Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Aleksi Peltonen

Formal Modelling and Verification of the EAP-NOOB Protocol

Master's Thesis
Espoo, July 31, 2018

Supervisor: Professor Tuomas Aura, Aalto University
Advisor: Mohit Sethi, D.Sc. (Tech.)

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author: Aleksi Peltonen

Title:
Formal Modelling and Verification of the EAP-NOOB Protocol

Date: July 31, 2018 **Pages:** viii + 82

Major: Computer Science **Code:** SCI3042

Supervisor: Professor Tuomas Aura

Advisor: Mohit Sethi, D.Sc. (Tech.)

The expansion of the Internet of Things (IoT) has resulted in an increasing number of new devices communicating independently over the network with each other and with servers. This has created a need for protocols to manage the swiftly growing network. Consequently, formal verification methods have become an important part of the development process of network systems and protocols. Before implementation, the specification itself has to be shown to be reliable and secure.

Nimble out-of-band authentication for EAP (EAP-NOOB) is a protocol for bootstrapping IoT devices with a minimal user interface and no pre-configured credentials. In this thesis, we create a symbolic model of the EAP-NOOB protocol with the mCRL2 modelling language and verify both its correct operation and its liveness properties with exhaustive state space exploration and model checking. Major findings relate to the recovery of the protocol after lost or corrupted messages, which could be exploited for denial-of-service attacks. We contribute to the standardisation process of the protocol by model checking the current draft specification and by suggesting improvements and clarifications to the next version. Finally, we verify the changes made to the protocol and show that they improve the overall reliability and fix the detected issues. Moreover, while modelling the protocol, we found various underspecified features and ambiguities that needed to be clarified. Furthermore, we create a test suite for testing the cryptographic implementation. By comparing message logs from the implementation with output generated by our test script, we find that incompatibilities between cryptographic libraries sometimes resulted in protocol failures.

Keywords: IoT, EAP, EAP-NOOB, mCRL2, formal verification, model checking

Language: English

Aalto-universitetet
Högskolan för teknikvetenskaper
Magisterprogrammet i data-, kommunikations- och informationsteknik

SAMMANDRAG AV DIPLOMARBETET

Utfört av:	Aleksi Peltonen		
Arbetets namn:	Formell Modellering och Verifiering av EAP-NOOB Protokollet		
Datum:	31 juli 2018	**Sidantal:**	viii + 82
Huvudämne:	Datateknik	**Kod:**	SCI3042
Övervakare:	Professor Tuomas Aura		
Handledare:	Mohit Sethi, D.Sc. (Tech.)		

Utvidgandet av sakernas internet (IoT) har resulterat i en ökning av nya fristående apparater som kommunicerar med varandra och med servrar. Detta har skapat ett behov av protokoll för att upprätthålla det växande nätverket. Följaktligen har användning av formell verifiering blivit en viktig del av utvecklingsprocessen av nätverkssystem och protokoll. Innan ett protokoll implementeras, måste själva specifikationen bevisas vara pålitlig och säker.

Nimble out-of-band authentication for EAP (EAP-NOOB) är ett protokoll för koppling av IoT-apparater med ett minimalt användargränssnitt och inga förhandskonfigurerade kreditiv. I detta examensarbete skapar vi en symbolisk modell av EAP-NOOB-protokollet med mCRL2 språket och verifierar diverse egenskaper genom tillståndsutforskning. Vi bidrar till protokollets standardiseringsprocess med förändringsförslag, visar att de förbättrar protokollets tillförlitlighet och korrigerar de upptäckta problemen. I samband med verifieringsprocessen hittade vi diverse tvetydigheter i specifikationen som korrigerades. Ytterligare presenterar vi ett testprogram för kryptografisk verifiering och datagenerering. Genom att jämföra loggfiler från implementeringen med våra genererade data visar vi att det existerar inkompatibiliteter mellan kryptografiska programbibliotek.

Nyckelord:	IoT, EAP, EAP-NOOB, mCRL2, formell verifiering, modell checking
Språk:	Engelska

Acknowledgements

First and foremost, I would like to thank Professor Tuomas Aura for accepting me to the EAP-NOOB protocol development team and for supervising my thesis. Throughout the project, Professor Aura has provided me with support, feedback and suggestions for improving my work. Furthermore, I would like to thank him for funding my trip to London to attend the IETF 101 meeting in March 2018.

I would like express my sincere gratitude to my advisor Mohit Sethi for his endless patience to answer my questions, help me with the project and give me feedback on my writing.

Finally, I would like to thank my family and friends for their moral support.

Espoo, July 31, 2018

Aleksi Peltonen

Abbreviations and Acronyms

ACK	Acknowledgement
BDD	Binary Decision Diagram
CTL	Computational Tree Logic
DoS	Denial-of-Service
EAP	Extensible Authentication Protocol
ECDH	Elliptic-Curve Diffie-Hellman
HML	Hennessy-Milner Logic
ID	Internet-Draft
IETF	Internet Engineering Task Force
Iff	If and Only If
IoT	Internet of Things
LPO	Linear Process Operator
LPS	Linear Process Specification
LTS	Labelled Transition Systems
MAC	Message Authentication Code
µCRL	micro Common Representation Language
mCRL2	micro Common Representation Language 2
MitM	Man-in-the-Middle
NAI	Network Access Identifier
NIST	National Institute of Standards and Technology
NOOB	Nimble Out-of-Band
OOB	Out-of-Band
P2S	Peer-to-Server
PBES	Parametrised Boolean Equation System
PINS	Partitioned Next-State Interface
PTL	Propositional Temporal Logic
RFC	Request for Comments
S2P	Server-to-Peer
TLS	Transport Layer Security
TTS	Timed Transition System

Contents

List of Figures

Chapter 1

Introduction

System development is an iterative process that consists of repeated cycles of specification, implementation, documentation and verification [5]. The specification of a system defines the behaviour of any implementation, whereas the documentation generally describes a specific implementation. The purpose of verification is to determine whether an implementation or specification has a set of properties, such as the desired functionality or the ability to recover from error situations. A system is considered to be correct if it satisfies all the specified properties [5]. These properties can range from generic requirements defined in the specification to implementation-specific features that describe the correct behaviour of a piece of code. Consequently, we can divide system verification into two categories: implementation (or software) verification and specification verification.

Traditional software verification techniques include testing and code reviewing, both of which have been proven to be effective ways to find defects in implementations [5]. Software testing is a dynamic verification technique that can be automated to verify a set of properties, expressed as pairs of matching input and output values, every time the implementation changes. After a change is made, each specified input is still expected to return the corresponding output. This technique is especially suitable for verifying software with a modular design, as it enables testing of individual parts, or modules, one at a time. However, testing can only show the absence or presence of pre-determined errors and it does not prove the overall correctness of a system [5]. Any property that is not explicitly defined by the developers will not be tested and mistakes made during the test implementation may generate false results. Furthermore, testing reactive systems creates additional challenges due to the fact that the execution does not always terminate. Many systems are meant to keep running without ever exiting, which makes verification with input-output pairs less applicable [17].

Code reviews solve these problems by not requiring execution of the software and help mitigate human error by involving more individuals in the process. After a feature passes all automated tests, it is reviewed by one or more developers, who are usually not directly involved in the development process of the feature. This way, mistakes made by one person are more likely to be detected before a feature is accepted for production and new errors that are not part of the automated test cases may be detected. Moreover, code reviews can improve the quality of software. Even if a piece of code works properly, it might be inefficient or redundant. Although an effective way to catch programming errors, manual reviewing can be time consuming and might still overlook subtle or obscure mistakes due to its dependence on humans [5].

Exhaustive verification becomes increasingly relevant when developing standards that serve as specifications for multiple independent implementations. In particular, the standardisation process of communication protocols often requires multiple iterations of improvements based on reviews and empirical examination. Prior to testing individual implementations with either of the aforementioned techniques, the protocol itself has to be verified to be correct in order to mitigate the possibility of design flaws. Any errors in the specification will likely propagate to the implementations. Correcting errors later requires modification of each individual implementation, which can include a significant amount of work and leave older versions vulnerable to failure until patches have been implemented and tested properly.

Formal verification is a group of techniques based on applied mathematics. These methods can be divided into two categories: deductive verification and model-based verification [5, 17]. Deductive verification includes inferring the correctness of a system specification with axioms and proof rules. It resembles mathematical proofs and requires a thorough understanding of the method to be applicable [17]. Model-based verification, on the other hand, involves creating a state model of the system and performing exhaustive exploration of all possible states. When an error state is reached, the model-checker typically provides a trace leading from the initial state to the error state. Formal verification methods are often used to prove the reliability of commonly used protocols, such as TLS 1.3 [40] (e.g. Cremers et al. [19]), and they have been used by companies such as Amazon [37] and Facebook [13] to eliminate bugs in large-scale services.

Symbolic modelling is a variant of model-based verification that strives to cover all relevant properties of a specification while abstracting away irrelevant details. Such details consist of data that does not affect the system behaviour [5]. The purpose of abstraction is to avoid the combinatorial explosion of system states [17]. The desired properties are expressed with a

property specification language, such as temporal logic, which allows referring to the sequence of actions or states occurring in the system over time [5]. This means that properties can include statements such as *the system will* ***eventually*** *reach state X*, or *the system will* ***never*** *reach state Y* ***after*** *reaching state X*.

The Internet Engineering Task Force[1] (IETF) is an organisation that maintains a collection of Internet specifications, called Requests for Comments (RFCs). Each RFC starts as an Internet-Draft (ID) and requires often multiple iterations of improvements before it is accepted as a standard. One such draft is the *nimble out-of-band authentication for EAP* (EAP-NOOB) [3], a bootstrapping protocol for Internet of Things (IoT) devices with minimal user interface.

EAP-NOOB is based on previous work by Sethi et al. [42, 43] and expands the Extensible Authentication Protocol (EAP) [45]. It has already undergone several iterations of improvements based on reviews of previous versions of the specification and the development of a software implementation [38, 41]. However, the protocol still requires additional iterations of feedback and improvements before standardisation. The main prospective verification tasks include assuring that the protocol design has the expected functionality, verification of linear properties, and understanding the protocol behaviour in various exceptional cases, such as error situations and after message loss.

The main objective of this thesis is to perform a thorough analysis of the protocol specification [2] with formal verification techniques. We will create a symbolic model of the current protocol draft with the formal specification language *micro Common Representation Language 2* (mCRL2). The model will be used to test the overall reliability of the protocol, as well as its ability to recover from errors and failures in the channels between servers and peers. Furthermore, we will simulate the protocol in a multi-server, multi-peer environment and verify that the desired properties are not affected by multiple simultaneous connections from different sources.

The goals of the research are as follows:

1. Create an mCRL2 model of the EAP-NOOB protocol.

2. Verify the correct behaviour of the protocol with automated queries and manual simulations. We will focus on verification of linear properties, i.e. correct behaviour of the client and server as reactive systems, as well as absence of denial-of-service situations.

[1] http://www.ietf.org/tao.html

3. Based on the results, provide updates to the protocol design team for the next version of the draft.

4. Perform software testing and targeted code reviews for the features that cannot be modelled with mCRL2, especially the cryptographic implementation.

Note that verification of the security of the cryptographic protocol is beyond the scope of this thesis and our focus is on verifying the correctness of the specification. For future work, see section 7.3.

1.1 Structure of the Thesis

The rest of this thesis is organized as follows. Chapter 2 describes the theoretical and historical background of model checking and presents the formal verification language mCRL2. Chapter 3 provides a brief introduction to the EAP-NOOB protocol and chapter 4 gives an in-depth explanation of how it was modelled. Chapter 5 presents the verification process by describing what was verified, how it was done, and what changes were made in the protocol specification and its implementation. Chapter 6 explains the software testing and code-reviewing techniques used for functional verification of the cryptographic implementation. Chapter 7 evaluates the modelling process, summarises the contributions that were made and discusses future work that could still be done. Finally, chapter 8 provides a summary of the thesis and concluding remarks.

Chapter 2

Model Checking Background

In this chapter, we present the historical and theoretical background of model checking. We begin with an explanation of the necessary background information by briefly introducing to the concept of a process algebra. After that, we describe the steps leading up to the development of the formal specification language mCRL2, used in this thesis for modelling the EAP-NOOB protocol, followed by an introduction to the language itself. We describe the concept of model checking and present its history and current state of art. Finally, we describe temporal logic as a way to verify models.

2.1 Process Algebra

In this thesis, we use symbolic modelling for verification of the reliability and correctness of the EAP-NOOB protocol. Symbolic modelling is a formal verification method based on *process algebra*, the study of parallel and distributed systems. A *process* describes the behaviour of a system, i.e. all events and actions that it can perform, including properties such as timing and probabilities. Alternatively, it can be viewed as a discrete event system, for which we can observe the occurrence of a behaviour at some moment in time. Modelling centralised, sequential systems can be done by examination of the processes as simple automata with finite numbers of states and transitions between them. The functionality of such models can be evaluated with a set of predetermined test cases of input-output pairs, simply by following the path from the initial state to a final state [4]. The downside of sequential models is their inability to model the interaction required by parallel or distributed (reactive) systems. Instead of a single execution path, the transitions of such systems might depend on the states of other parts running in parallel, possibly on separate hosts.

The aim of a *process algebra* is to solve the problem of sequential models by providing means of performing calculations with processes, describing distributed systems and specifying interactions between them. Furthermore, it allows defining alternative and sequential composition, i.e. choice and sequential behaviour [4]. The basic laws of process algebra are the structural laws concerning the basic operations + (nondeterministic choice) and . (sequential composition). Finite processes can be generated with five basic axioms [7]. In addition to these, additional theorems define the behaviour of the merge, abstraction, and communication operators [4].

Axiom 1 *Commutativity of nondeterministic choice:*
$x + y = y + x$

Axiom 2 *Associativity of nondeterministic choice:*
$x + (y + x) = (x + y) + x$

Axiom 3 *Idempotence of nondeterministic choice:*
$x + x = x$

Axiom 4 *Right distributivity of nondeterministic choice over sequential composition:*
$(x + y) \,.\, z = x \,.\, z + y \,.\, z$

Axiom 5 *Associativity of sequential composition:*
$(x \,.\, y) \,.\, z = x \,.\, (y \,.\, z)$

Model checking requires both nondeterministic choice and sequential composition in order to generate *all* possible execution paths, or futures, for exhaustive state space exploration. For example, as shown in listing 2.1, an in-band server-to-peer channel is given the choice of receiving and sending either a success message (row 3) or a failure message (row 6). Additionally, instead of forwarding the message, the channel can drop it (rows 4 and 7) and raise an error to notify the user that the message was not delivered. This way, whenever the channel receives a message, it creates two distinctive paths to explore: one in which the message is delivered and one in which it is dropped. Both cases should be considered by the verification process to cover all possible execution paths for messages sent through the channels.

```
% In-band channel: Server to Peer
ServerToPeerChannel() =
    SEND_EAP_SUCC_I
    . (RECV_EAP_SUCC_O + ERROR_MSG(dropped_msg))
    . ServerToPeerChannel()
    + SEND_EAP_FAIL_I
    . (RECV_EAP_FAIL_O + ERROR_MSG(dropped_msg))
    . ServerToPeerChannel()
;
```

Listing 2.1: In-band server-to-peer channel

Algebra of Communicating Processes (ACP) is a process algebra developed to investigate the solutions of unguarded recursive equations [4]. It was first introduced by Bergstra and Klop [7] in a paper which also included the original definition of the term *process algebra*. In addition to the basic laws (A1-A5), ACP defines two axioms for the deadlock action [7]:

Axiom 6 *Deadlock 1:*
$\delta + x = x$

Axiom 7 *Deadlock 2:*
$\delta \cdot x = \delta$

The deadlock action δ was included to describe failure of a process; no other action can take place after it, forcing the process to terminate unsuccessfully [7].

2.1.1 micro Common Representation Language 2

The *micro Common Representation Language* (µCRL) was developed by Groote and Ponse [24] for studying processes with data. It is based on previous work by Bergstra [7] and extends ACP with the notation of data. µCRL was designed to be as simple as possible, while still capable of modelling realistic systems. However, due to its simplistic design, it lacked some of the basic functionality required to describe complex systems. During the decade following its initial release, µCRL went through various major changes, such as being extended with time, constructors and the ability to choose the initial state.

The toolset translates a given µCRL-specification to a *linear process operator* (LPO), which can be used for simulation or translated into an optimised *labelled transition system* (LTS) [10], as shown in figure 2.1.

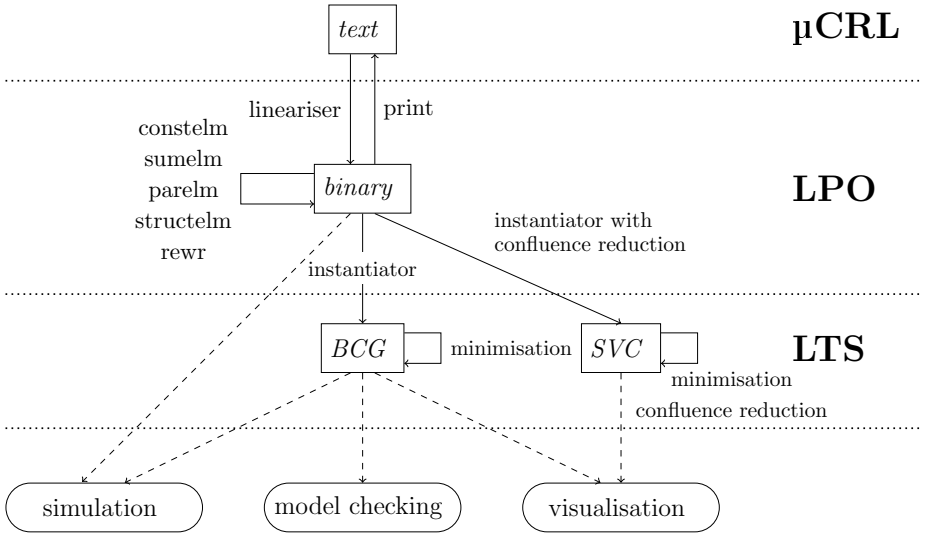

Figure 2.1: µCRL Overview [10]

µCRL was later replaced by its successor, mCRL2, which further improved the notation of data with higher-order abstract data types. The new version introduced standard data types, such as sorts, natural numbers, function types, structured types, lists, sets and bags. Additionally, it includes support for multi-actions (combined collections of more than two actions) and operators for blocking, allowing, and defining communication of actions [22, 23]. Similar to its predecessor, mCRL2 was developed for analysing complex, distributed and parallel systems. It is a collection of tools for simulation, minimisation, visualisation and model checking. The language itself consists of three sub-languages [18]:

1. The **data language** expresses data with abstract data types, such as Booleans and natural numbers. Other, non-default data types can be constructed with type constructors, such as sets, lists and functions.

2. The **process language** describes the behaviour of a system. Each system consists of one or more processes that are composed of user-defined operators, such as multi-action composition and abstraction operators.

3. The **property language** describes high-level temporal properties. It is an extension to modal µ-calculus that adds the notion of data, allowing queries to include variables.

The mCRL2 toolset is a collection of over 60 tools for analysing a system expressed with the specification language [18].

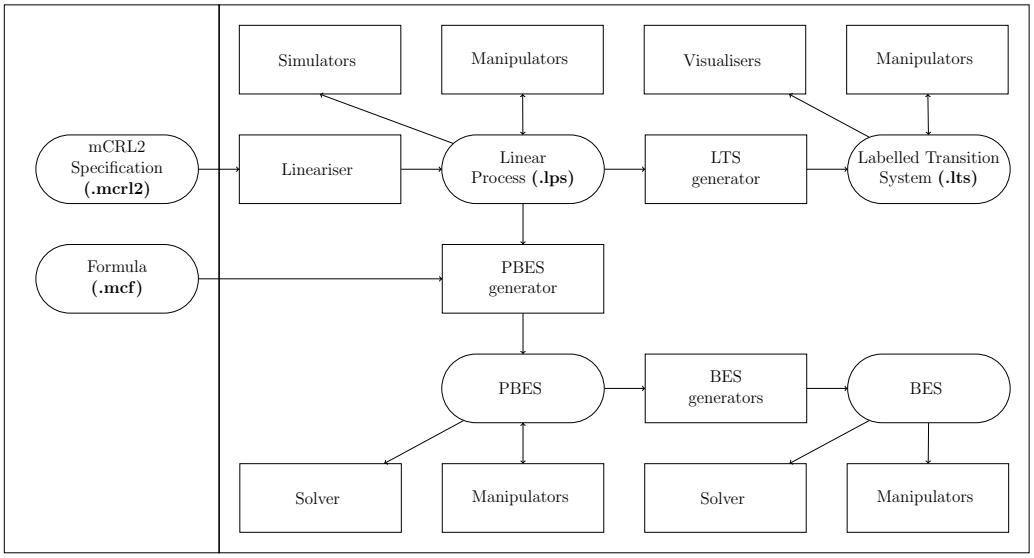

Figure 2.2: mCRL2 Toolset [44]

An mCRL2 process specification (*.mcrl2*) is initially transformed into a *linear process specification* (LPS), removing any parallelism and reducing the complexity of the processes with behaviour-preserving transformations [18]. The LPS is finite and can be used for simulation, reduction and viewing statistical information. Furthermore, it can be optimised into a *labelled transition system* (LTS), or state space, which can be visualised as a state transition graph or a 3D model [21].

A labelled transitions system LTS is defined as follows [23] :

$$LTS = (S, ACT, \rightarrow, s, T) \tag{2.1}$$

where S is a set of states, ACT is a set of actions, $\rightarrow \subseteq S \times Act \times S$ is a transition relation, $s \in S$ is the initial state, and $T \subseteq S$ is the set of terminal states.

For example, the LTS $L = (\{s_0, s_1\}, \{A\}, \{(s_0, A, s_1)\}), \{s_0\}, \{s_1\})$ represents the transition system depicted in figure 2.3.

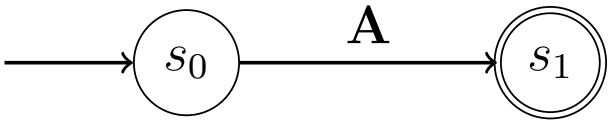

Figure 2.3: Visualisation of an LTS as a State Transition Graph

2.2 Model Checking

Model checking is a verification technique based on the formal specification of a system. The examined problem is as follows: given a model M and a formula f, find states which satisfy f. Formally, a model M is defined as [17]:

$$M = (S, S_0, R, L) \tag{2.2}$$

where S is a finite set of states, $S_0 \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is a binary relation of transitions, and $L : S \to 2^{AP}$ is a set of every atomic proposition (AP).

A model checker typically consists of three parts [15, 16]:

1. A **property specification language** (usually based on a temporal logic) for describing the required *properties*.
2. A **model specification language** for describing the *system* to be verified.
3. A **verification procedure** for state space exploration to verify the specified properties. In other words, for each formula f, check if it is satisfied by the model M. If not, provide a counterexample that violates the truthfulness of the statement.

The properties are generally expressed with some temporal logic that allows describing transitions in terms of time, often without explicitly defining it. This allows formalising statements such as *proposition p holds* ***sometime*** *in the future*, *proposition p holds* ***always*** *in the future*, or *proposition p holds* ***until*** *proposition q holds*. Many variations of temporal logic have been created, mainly differing in the way operations are expressed [5, 17].

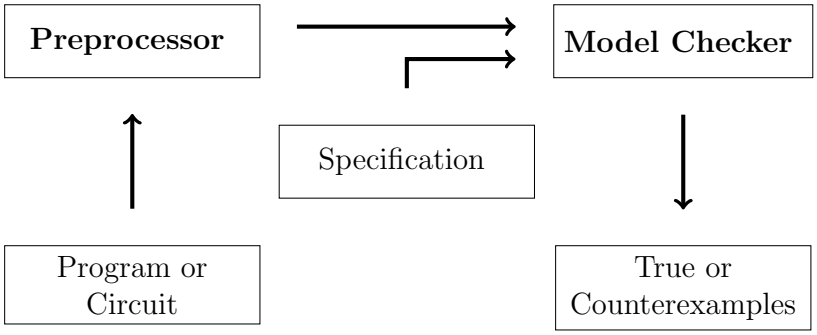

Figure 2.4: Overview of a Model Checker [16]

The model checking process consists of three steps [5]: (1) the *modelling* step, (2) the *running* step, and (3) the *analysis* step. In the first phase, the system is modelled with some specification language and the required properties are formalised. In the second phase, the formalised properties are tested in the state space of the created model. Falsification of a property is often faster than verification, since finding a single counterexample suffices to prove that it does not hold globally. On the other hand, if no counterexamples were found, the entire state space needs to be explored to show that there are in fact no counterexamples to be found [15]. Finally, in the third phase, the model, the design and the properties are refined according to results from the previous phase [5].

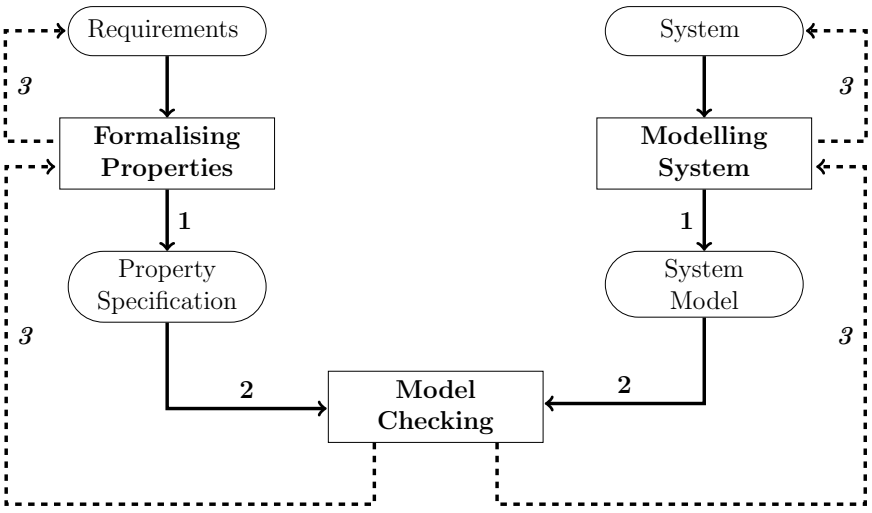

Figure 2.5: The Model Checking Process

The two main challenges that model checking tries to solve are the *model validation problem* and the *verification problem*. The model validation problem asks whether a model and the given properties describe a system with sufficient precision, whereas the verification problem asks whether a system is specified or implemented correctly. An incorrect or insufficient model may give false positives or negatives and thus be completely useless for the refinement process [5]. When verifying a system with modelling, three main types of errors may occur: (1) modelling errors happen when the implemented model does not describe the system correctly (validation problem), (2) design errors when the design of the system itself is faulty and needs to be changed, and (3) property errors when a formalised property does not correctly describe a requirement [17].

The main strengths and weaknesses of model checking according to Baier and Katoen [5]:

+ Model checking is a general verification approach that can be integrated to any stage of the system design and implementation process. It supports partial or complete verification and provides means of verifying that a protocol is correct before any actual software implementation is made. Finding bugs in early stages is a major step towards correct and secure systems and will often save a significant amount of time and other resources.

+ The verification process itself is often completely automatic. The main task of the developer is to create correct and complete models of the systems that need to be verified. As with traditional testing, the additional properties can be formalised and re-tested any time the model itself is changed.

+ Model checking does not depend on the likelihood of an error occurring, because the entire state space is examined. Formalising properties with temporal logic helps to find obscure mistakes that might be exploited by malicious parties.

+ Many modelling tools provide diagnostic information, such as execution traces, when problems are discovered. This information can be helpful when reproducing the failures in the actual implementations in order to fix the issues.

− Model checking verifies the specification, not the actual implementation. Even though the model is proves the correctness of the specification, any implementation might contain bugs or be dependent on other vulnerable software. Furthermore, model checking only verifies the stated properties. Anything that is not explicitly formalised as a property will not be tested.

− The model checker itself might contain bugs and give false results, even if the model is correct and complete. Thus, the verification software needs to be thoroughly tested in order to avoid false positives and negatives.

− Model checking is generally not suitable for data-intensive applications and suffers from the state-explosion problem. If the amount of data that cannot be abstracted away grows too large, the verification process might end up taking an infeasible amount of time or memory.

The state-explosion problem is a consequence of the combinatorial growth of system states. This issue is often likely to occur during the model checking process, since it relies on exhaustive search of the state space to verify properties. In order to avoid a state-explosion, Clarke et al. [17] list four main categories of countermeasures:

1. *Abstraction* of irrelevant data is the main approach to minimise the number of states throughout the modelling process. In cases where the actual content of some data does not affect the behaviour of the system, it can be abstracted to a smaller set of values, such as undefined integers or strings. This removes unnecessary states and transitions that ultimately are exactly the same as each other.

2. *Partial order reduction* involves dropping sequences that are impossible to distinguish between when disregarding the ordering of independent events. In other words, if two separate paths with a common starting point lead to the same results, only one of them needs be considered in the verification process.

3. *Compositional reasoning* for modular structures divides the system into sub-parts that can be independently checked and shows that correct behaviour of the components implies the correct behaviour of the entire system. Although efficient, this approach requires the system and the modelling method to be modular in order to be applicable.

4. *Symmetry* can be exploited to get rid of actions that do not affect the state of the system. This is done by removing non-trivial permutations that preserve the state transition graph.

Furthermore, symbolic model checking is often applicable for verifying large systems (e.g. Burch et al. [12], Biere et al. [8]). The idea of a symbolic model is to avoid explicit state representations, which can be achieved by exploiting the regularity of the state space to represent it symbolically with e.g. a *binary decision diagram* (BDD) [12].

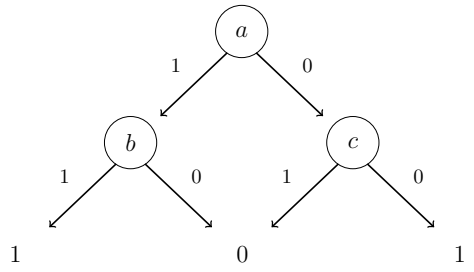

Figure 2.6: BDD representation of $(a \wedge b) \vee (\neg a \wedge \neg c)$

2.2.1 History of Model Checking

Model checking based on temporal logic was first introduced by Clarke and Emerson [14] as an alternative to constructing proofs, without axioms and rules [17]. The approach suggests that a system can be modelled as a finite state machine and the desired properties as clauses with a *propositional temporal logic* (PTL). The (bounded) finite model property for PTL states that if a formula f is satisfiable, it is satisfiable in a finite model, and a finite model of f can be constructed. Furthermore, in a finite state model, each required property, expressed as a PTL formula, can be tested against the (finite) state space of the model [17].

In addition to introducing the concept, Clarke and Emerson [14] describe a model checker for deciding whether a finite structure models a given formula. The verification process searches the entire state space to decide whether the clauses hold, with a high-level temporal logic specification to automatically produce a synchronisation skeleton of the system; an abstraction with irrelevant details removed. The original version of the verification tool was designed for concurrent programs in a shared-memory environment, although it can be extended for distributed systems. Finally, Clarke and Emerson [14] present a model-checking algorithm for verifying existing (finite state) programs.

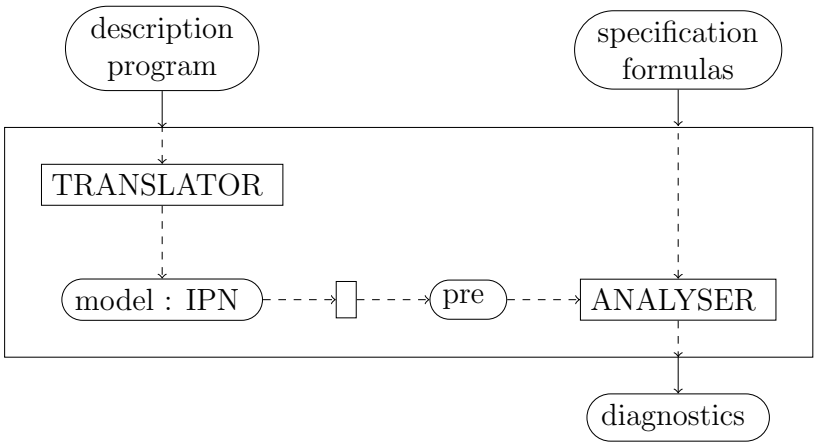

Figure 2.7: CESAR [39]

In an unrelated paper, Queille and Sifakis [39] introduce *CESAR*, a model checker for concurrent, distributed systems. A concurrent system consists of a finite number of processes $\{P_1, \ldots, P_m\}$. Each process can be described with a flow graph that consists of nodes that represent states in the system and connections between nodes that show transitions from one state

to another [39]. Much like the model checker introduced by Clarke and Emerson [14], *CESAR* validates the description of a system, expressed in a high-level language. The verification process works by abstracting away irrelevant details, such as data, and focuses on testing the control structure [39]. Properties to be tested are given as a set of specification formulas, expressed in branching time logic, and consist of invariant and liveness properties, as well as properties of response to an action.

Throughout the history of model checking (with state exploration techniques) the *state-explosion problem* [16] has been one of the main challenges to overcome. As reactive systems have grown larger in size, modelling real-life systems has become increasingly difficult. Most of these modern systems are practically impossible to perform exhaustive searches on and many of them have an infinite amount of states if no abstraction is applied. However, the importance of modelling as a verification tool has increased due to an increase in safety-critical systems depending on both software and hardware. Furthermore, in recent years, the trend of concurrent, distributed systems has created new challenges for modelling tools to keep up with [15].

In addition to improving the modelling tools, two major trends have emerged to accompany the model-checking research: *synthesis* and *model measuring*. Whereas verification is concerned with ensuring that a system matches its specification, synthesis involves the task of constructing the system based on a specification. Furthermore, model measuring tries to decide how well a system matches the specification. Synthesis aims to eliminate any errors in the implementation phase and model measuring allows defining an acceptable error margin, a feature that most traditional model checking approaches lack [15].

2.3 Temporal Logic

A model checker typically consists of a verification procedure and two specification languages: one for formalising the model itself and one for formalising the required properties under test. The property specification language is generally based on a temporal logic, which makes it possible for statements to include time as a constraint [15, 16]. There are multiple variants of temporal logic that differ in both syntax and semantics. In this section, we provide an overview of three notable logics in regards to our work: CTL*, Hennesy-Milner logc (HML) and modal µ-calculus.

2.3.1 CTL*

CTL* is an extension to computational tree logic (CTL) that combines it with features from linear temporal logic (LTL) [5].

A CTL* *state formula* is defined over the set of atomic propositions AP:

$$\phi ::= true \mid a \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \mathrm{E}\varphi. \tag{2.3}$$

where $a \in AP$ and φ is a CTL* *path formula*:

$$\varphi ::= \phi \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathrm{X}\varphi \mid \varphi_1 \mathrm{U} \varphi_2. \tag{2.4}$$

where X (next) and U (until) are temporal modalities.

In addition to the basic syntax, the U (until) operator is often used to derive two additional temporal modalities:

$$\mathrm{F}\phi := true \mathrm{U}\phi \ \ (finally) \quad \ and \quad \ \mathrm{G}\phi := \neg\Diamond\neg\phi \ \ (globally) \tag{2.5}$$

With these additions, model checking reachability in a state space becomes relatively straight forward. For example, we can formulate the absence of deadlocks simply as: *some action can **globally** happen*, or reachability of a given state as: *the system will **finally** reach the desired state.* The universal path quantifier A can be defined as follows:

$$\mathrm{A}\varphi = \neg\exists\neg\varphi \tag{2.6}$$

2.3.2 Modal μ-Calculus

Hennessy-Milner logic [26, 27] is an alternative to CTL* that specifies reachability with the diamond and box modalities. The diamond modality $\Diamond_a\phi$ is valid whenever an action a can be performed s.t. ϕ holds afterwards. Correspondingly, the box modality $\Box_a\phi$ is valid iff for every action a, ϕ holds afterwards [23].

The syntax of HML is as follows [23]:

$$\phi ::= true \mid false \mid \neg\phi \mid \phi \ \wedge \ \phi \mid \phi \ \vee \ \phi \mid \phi \rightarrow \phi \mid \Diamond_a\phi \mid \Box_a\phi. \tag{2.7}$$

where ϕ is a formula, $\wedge$ (and), $\vee$ (or) and $\neg$ (not) are Boolean connectors, $\rightarrow$ is implication, a is an action, $\Diamond$ is the diamond modality, and $\Box$ is the box modality.

Standard HML provides enough flexibility to express both finite and infinite systems but lacks expressiveness in many areas. Mainly, it does not support representing data or specifying fairness properties (*=given an infinite amount of chances to perform an action, it must eventually occur*). Modal µ-calculus [33] extends HML with the minimal and maximal fix point operators [23]:

- the *minimal* fix point operator μ refers to the *smallest* set that satisfies an equation (e.g. $\mu X.\phi$, where X is a set of states)
- the *maximal* fix point operator ν refers to the *largest* set that satisfies an equation (e.g. $\nu X.\phi$, where X is a set of states)

The definition of a formula ϕ in µ-calculus includes the definition of standard HML, as well as the extended operators μ and ν [23]:

$$\phi ::= true \mid false \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \Diamond_a\phi \mid \Box_a\phi \mid \mu X.\phi \mid \nu X.\phi \mid X. \quad (2.8)$$

where $\wedge$ (and), $\vee$ (or) and $\neg$ (not) are Boolean connectors, $\rightarrow$ is implication, a is an action, $\Diamond$ is the diamond modality, $\Box$ is the box modality, X is a predicate variable, and μ and ν are the minimal and maximal fix point operators.

The mCRL2 language includes a first-order modal µ-calculus (extended with data) for formalising the requirements of the system [23]. These requirements, expressed as properties, are tested by translating the corresponding formula and process into a parametrised boolean equation system (PBES) that consists of minimal and maximal fix point equations [18]. The solution to a generated PBES determines whether or not the property holds for the process.

The typical form of a single equation is as follows:

$$(\mu X(d : D) = \phi) \text{ or } (\nu X(d : D) = \phi) \quad (2.9)$$

where X is a predicate variable, d is a formal variable of type D, ϕ is a predicate formula, and μ and ν are the minimal and maximal fix point operators.

Chapter 3

EAP-NOOB

In this chapter, we provide an overview of the EAP framework and the EAP-NOOB method. We focus on explaining the structure of the state machine and describe the general concept of exchanges in the EAP-NOOB protocol. Further details about individual exchanges and the key derivation process can be read in the current protocol draft [3].

3.1 EAP Framework

The Extensible Authentication Protocol (EAP) [45] is a flexible framework for authentication methods. The purpose of it is to provide reliable transportation of parameters for key generation. As the name suggests, EAP itself is not a complete authentication mechanism, but rather a framework for authentication methods, such as EAP-NOOB. The three main parties present in an EAP exchange are the peer, the authenticator, and the authentication server (see figure 3.1). The authentication procedure begins with a request from the authenticator to the peer, which the peer counters with an appropriate response. Depending on the method, the authenticator will either process the response and possibly send further requests to the peer or operate as a pass-through node, forwarding the packets between the peer and an authentication server.

An EAP packet consists of four or more octets of data and contains the following information: a type code identifying the packet type, an identifier for matching it with an appropriate request or response, the length of the packet, and optional data. The supported packet types are request, response, success, and failure. Each exchange consists of one or more request-response pairs and always ends with either a success (if the method was completed) or a failure.

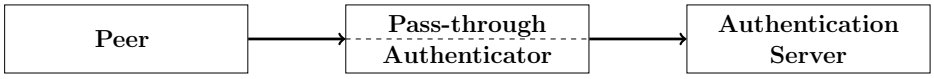

Figure 3.1: EAP Framework Overview

3.2 EAP-NOOB Protocol Overview

Nimble out-of-band authentication for EAP (EAP-NOOB) [3] is an EAP method for bootstrapping IoT devices with minimal user interface and no preconfigured authentication credentials. It requires either input (e.g. cameras) or output (e.g. displays) from the peer device and creates a set of keys for secure communication between a device and a server. The authentication process is based on a user-assisted one-directional out-of-band (OOB) channel. The EAP-NOOB state machine contains three ephemeral (states 0-2) and two persistent (states 3-4) states. Initially, both parties start out in the Unregistered (0) state with no exchanged values. The pairing process consists of four EAP exchanges and one user-assisted step. Each exchange delivers a set of values to the other endpoint and negotiates supported versions and other variables. After reaching one of the persistent states, only a user reset can revert the state of a device or server back to an ephemeral one. In case of data loss (failure or reset), a peer is treated as a new device with no previously exchanged values.

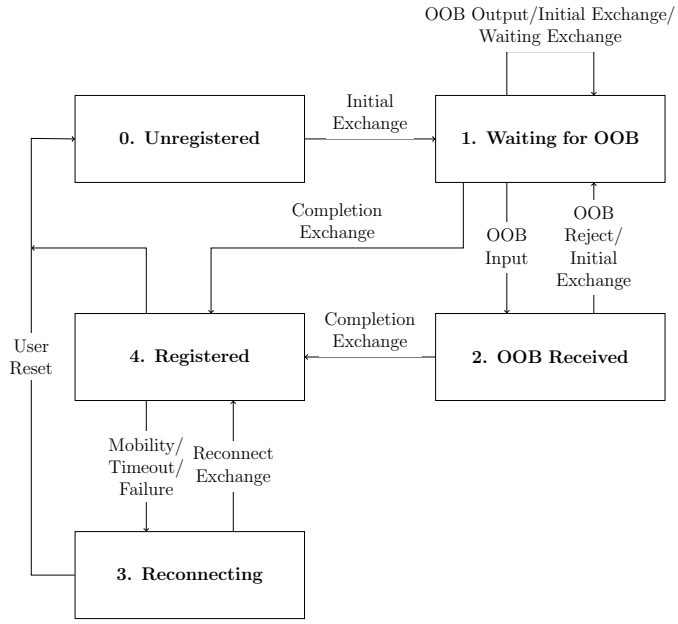

Figure 3.2: EAP-NOOB State Machine [3]

Exchange	Purpose
Initial Exchange	- The server allocates a unique identifier (*PeerId*) to the Peer - Protocol and cryptosuite versions are negotiated - Nonces are exchanged - ECDH parameters for key derivation* and MAC** calculation are exchanged
Waiting Exchange	- The server informs the peer that the OOB step is not yet complete and the Completion Exchange cannot yet be initiated
OOB Step	- One of the parties creates a secret nonce *Noob* and sends it to the counterpart, together with a cryptographic fingerprint *Hoob*
Completion Exchange	- Mutual authentication is completed - Keys are confirmed
Reconnect Exchange	- Protocol and cryptosuite versions are re-negotiated - Nonces are exchanged - ECDH parameters for re-keying* and MAC** calculation are exchanged

* = Elliptic Curve Diffie-Hellman (ECDH) algorithm following the NIST specification [6]
** = Computed with the HMAC [34] function

Table 3.1: EAP-NOOB Exchanges

An exchange is always initiated by the peer (with the exception of a server-to-peer OOB message) after receiving an EAP-Reqeuest/Identity packet from the authenticator (an entity initiating the EAP authentication). The peer responds by sending out a network access identifier (NAI) [20] that consists of the concatenation of a unique peer identifier (*PeerId*) allocated by the server during the initial exchange ("noob" for unregistered devices), the current state (X) of the peer (unless the state is 0), and the suffix *@eap-noob.net*. The server chooses the appropriate exchange based on the states of the server and the peer. The correct exchanges for each pair of states are listed in table 3.2.

Server state	Peer state*	Exchange
0	0 1 2	Initial Exchange
1	0 1 2	Initial Exchange Waiting Exchange Completion Exchange
2	0 1 2	Initial Exchange Waiting Exchange Completion Exchange
3 4	3	Reconnect Exchange

* = An exchange should never be initiated when the peer is in the Registered (4) state.

Table 3.2: Exchange Chosen by Server

The rest of the exchange consists of one or more EAP-Request/EAP-NOOB and EAP-Response/EAP-NOOB messages, and ends with an EAP-Failure or EAP-Success. Each request/response contains a payload and can be identified by the receiver by its type parameter (Type $\in \{0 \dots 8\}$).

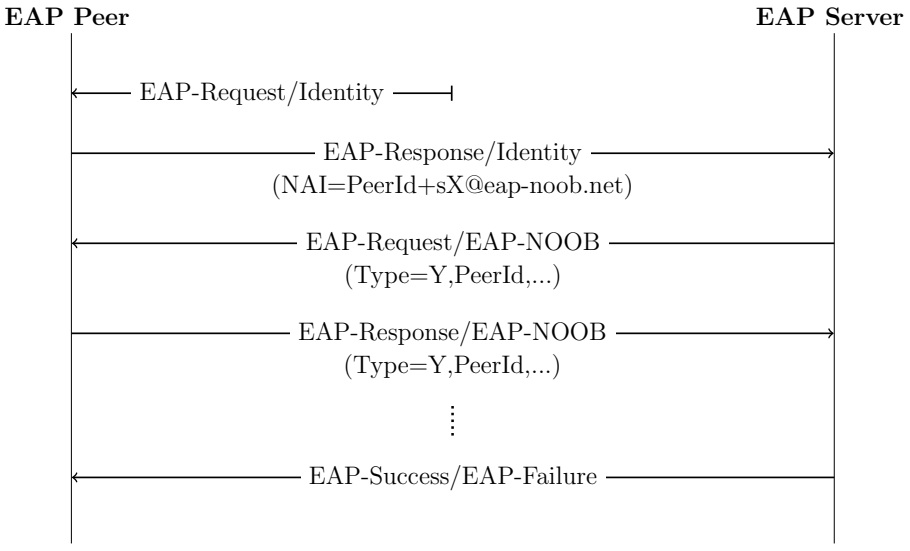

Figure 3.3: EAP-NOOB Exchange

An exception to the described exchange pattern is the user-assisted OOB step that occurs once or more before the Completion Exchange. The direction

of OOB messages is negotiated during the Initial Exchange, and has three alternatives: (1) peer-to-server (P2S), (2) server-to-peer (S2P), or (3) both P2S and S2P are allowed and the first message to be delivered is accepted. The OOB step requires the assistance of the user and can for example consist of scanning a bar-code with a camera. It delivers a 16-byte secret nonce *Noob* and a 32-byte cryptographic hash value *Hoob*, computed from the previously exchanged values. The accepting party computes the corresponding hash value *Hoob* and only proceeds to the next step if the hashes match and if the nonce is still valid. In case of multiple delivered OOB messages the receiver can choose which one to accept.

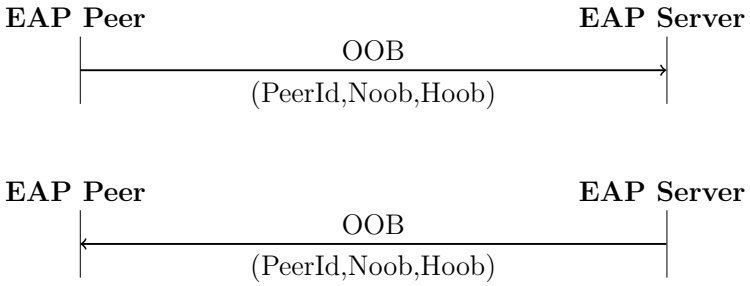

Figure 3.4: P2S/S2P OOB Step

Any errors occurring during an exchange can either lead to an error recovery procedure or to the peer informing the user that recovery is impossible. Recoverable errors may happen due to non-persistent reasons, such as packet loss or modification in the channel. Other errors, such as mismatching version numbers of supported protocols, need to be solved manually by the user. The error handling procedure starts with a message of type 0, followed by an EAP-Failure from the server.

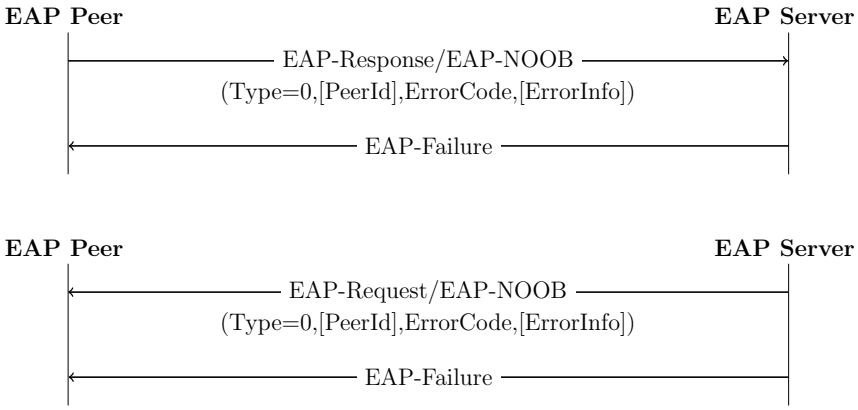

Figure 3.5: P2S/S2P Error Reporting

The correct behaviour after receiving an error message depends on the ongoing exchange:

- During the **Initial Exchange**, both parties revert back to the Unregistered (0) state and start over.
- During the **Waiting/Completion Exchange**, both parties remain in their current state but abort the ongoing exchange.
- During the **Reconnect Exchange**, both parties change their state to the Reconnecting (3) state and abort the ongoing exchange.

Chapter 4

Modelling EAP-NOOB

In this chapter, we describe the modelling process of the EAP-NOOB protocol. We begin with presenting an overview of the model itself, followed by a description of how data, communication and storage is represented in it. Finally, we discuss how time and timing properties are modelled.

4.1 Model Overview

The model (see Appendix B) we created for this thesis represents the current version of the EAP-NOOB protocol [3] (draft version 03) with some minimisation techniques (mainly abstraction and partial order reduction) applied in order to reduce the state space. It consists of three main processes: (1) the server, (2) the peer, and (3) the server-side database. In addition to these, two sub-processes are included to manage nonce generation and error handling, respectively.

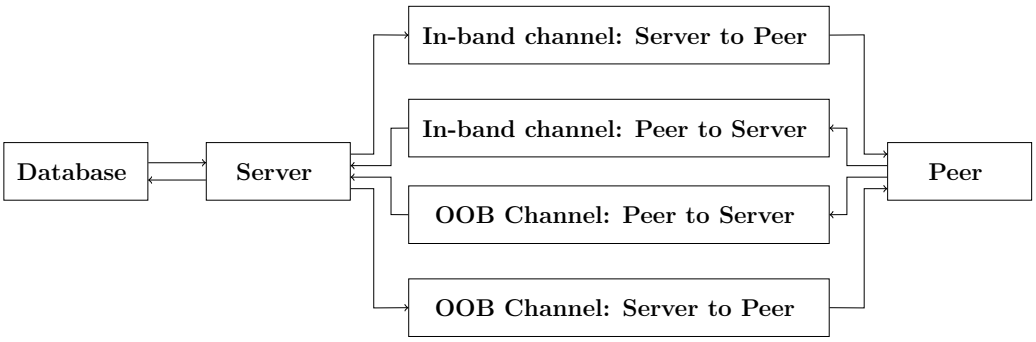

Figure 4.1: EAP-NOOB Model Overview (one server, one peer)

Each specified process can be spawned one or more times depending on the attributes under test. The decision of how many servers and peers to include in an instance of the model is made before the compilation and needs to be defined at the end of the model specification file (*eap-noob.mcrl2*). For example, one server can be created for communicating with multiple peers, each of which is recognisable by a unique *PeerId*. All communication between the processes is defined as pairs of synchronised actions that create atomic multi-actions. Every *PeerId* recognised by the server, along with exchanged data, is stored in the server-side database. When a peer initiates a connection to the server, the peer-specific data is fetched from the database and subsequently updated.

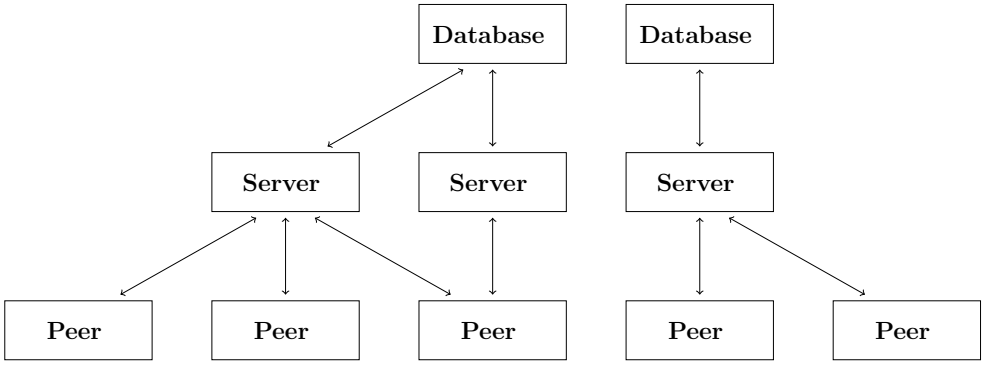

Figure 4.2: EAP-NOOB Model Overview (multiple servers, multiple peers)

Furthermore, the model contains a process that represents a pseudo-random value generator, which creates unique *PeerIds* and nonces (both represented as natural numbers). To avoid a state-explosion, the number of generated values must be limited depending on the features under test. The minimum number of values needed for each state to be reachable is one *PeerId*, two *Noob* values (one if sending the OOB message is only allowed in one direction: S2P or P2S), and four nonces.

```
% Maximum number of PeerIds, Noob values, nonce values in
% the model
max_peers  = 1;
max_noobs  = 2;
max_nonces = 4;

% Maximum number of failed OOB messages before going back to
% state 0 (Unregistered)
max_oob_retries = 2;
```

Listing 4.1: Limiting the model to avoid unnecessary states

4.1.1 Data Types

The EAP-NOOB protocol requires the servers and peers to validate and use the exchanged data. Each exchange consists of sending and receiving device-specific values that are stored and later compared by calculating message authentication codes (MACs) with the HMAC [34] function. If the HMAC does not match the previously exchanged values, the message in which it was sent in is rejected. As a result, most of the data cannot be abstracted away and needs to be assigned comparable values. To achieve this as cost-effectively as possible, we substitute most data types with a natural number (*Nat*) or a list of natural numbers.

```
% Data types
Ver_t          = Nat;                  % Version
Ver_l          = List(Ver_t);          % List of versions
Cryptosuite_t  = Nat;                  % Cryptosuite
  ⋮

NoobId_t       = Nat;                  % Noob ID
```

Listing 4.2: Mapping of data types to natural numbers

The values that cannot be abstracted away or represented using only numbers are defined as custom data types. mCRL2 allows to define new types as comparable structures, which consist of values and other custom types. For example, we represent a derived key with a structure that contains the parameters used for the key-derivation. The same technique is used for modelling MACs, packets, and other complex data types.

```
% Cryptographic MAC
MAC_t = struct HMAC(K:K_t,
     Dir:Nat, Vers:Ver_l, Verp:Nat, PeerId:Nat,
     Cryptosuites:Cryptosuite_l, Dirs:Nat, ServerInfo:Info_t,
     Cryptosuitep:Nat, Dirp:Nat, PeerInfo:Info_t, PKs:Nat,
     Ns:Nat, PKp:Nat, Np:Nat, Noob:Nat)
;
% Derived keys
K_t = struct no_key
   | Completion(slice:Nat, Z:ECDH_t, Np:Nat, Ns:Nat, Noob:Nat)
   | RekeyingECDH(slice:Nat, Z:ECDH_t, Np2:Nat, Ns2:Nat, Kz:K_t)
   | Rekeying(slice:Nat, Kz:K_t, Np2:Nat, Ns2:Nat)
;
```

Listing 4.3: Custom MAC and key structures in the model

```
% Data sent between peer and server
Data_t = struct empty_d ? is_empty
    % Type 1-8 requests and responses
    | req1(Vers:Ver_l, Cryptosuites:Cryptosuite_l, Dirs:Nat,
      ServerInfo:Info_t) ? is_req1
    | res1(Verp:Nat, Cryptosuitep:Nat, Dirp:Nat, PeerInfo:
      Info_t) ? is_res1
    | req2(PKs:Nat, Ns:Nat) ? is_req2
    | res2(PKp:Nat, Np:Nat) ? is_res2
    | req3 ? is_req3
    | res3 ? is_res3

    ⋮

    | req8 ? is_req8
    | res8(NoobId:Nat) ? is_res8
;
```

Listing 4.4: Custom data structure in the model

4.1.2 Communication

The model specifies four types of channels: two for communicating from a peer to a server (one in-band, one out-of-band), and two for communicating from a server to a peer (one in-band, one out-of-band). All communication between the servers and peers happens over these channels. Each server-peer pair has one out-of-band channel in each direction for the OOB messaging and one in-band channel in each direction for the network communication. Sending a message over any channel consists of four actions (I = in, O = out, * = message type):

$$\textbf{Sender} \xrightarrow[1]{\text{(SEND\_*\_O)+(SEND\_*\_I)}}_{2} \textbf{Channel} \xrightarrow[3]{\text{(RECV\_*\_O)+(RECV\_*\_I)}}_{4} \textbf{Receiver}$$

Figure 4.3: Message Delivery in the Model

1. The message is sent to the channel
2. The message is received by the channel
3. The message is sent to the receiver
4. The message is received by the receiver

The supported message types in the in-band channels are REQ (EAP-Request/EAP-NOOB), RES (EAP-Response/EAP-NOOB), RES_ID (EAP-Response/Identity), FAIL (EAP-Failure) and SUCC (EAP-Success). Correspondingly, the supported message types in the out-of-band channels are OOB_P2S and OOB_S2P. The first two actions are synchronised with a multi-action, as are the two last ones: *_O and *_I happen simultaneously, or not at all.

```
% OOB Channel: Peer to Server
PeerToServerOOBChannel =
    sum PeerId:PeerId_t, Noob:Noob_t, Hoob:Hoob_t . (
        SEND_OOB_P2S_I(PeerId, Noob, Hoob)
        . RECV_OOB_P2S_O(PeerId, Noob, Hoob)
        . PeerToServerOOBChannel
    )
;
```

Listing 4.5: P2S OOB channel

However, as shown in listing 2.1, messages may be dropped by the in-band channels. After a channel has received a message, it can choose to either deliver it to its destination or simply drop it and continue to wait for new packets. The participants are not informed of the dropped message and are responsible for detecting and correcting any consequent errors. In practice, this means that two distinctive execution paths will be created for every packet: one where the message is dropped and one where it is delivered.

Furthermore, the in-band channels can be compromised by an attacker, which may modify messages before the channel delivers them. As the protocol itself has no way of resisting against persistent denial-of-service attacks, the modelled attacker is limited to a certain number of spoofed messages before leaving the channel. Examining the protocol behaviour with compromised channels verifies that issues caused by invalid data, such as spoofed nonces or peer identifiers, do not lead to persistent error states.

4.1.3 Server-Side Database

The server-side database is a process responsible for storing peer data for each individual *PeerId* recognised by the server. Whenever a server communicates with a peer, any peer-specific values are fetched from the database and after the interaction, the values are stored or updated as needed. The database is implemented as a key-value structure and provides an interface to the server for reading and writing data.

```
% Association database entries (keyed by PeerId:Nat)
State_e         = PeerId -> State_t;         % Current state
Verp_e          = PeerId -> Verp_t;          % Version
Cryptosuitep_e = PeerId -> Cryptosuitep_t; % Cryptosuite
Dirp_e          = PeerId -> Dirp_t;          % OOB Direction
PeerInfo_e      = PeerId -> Info_t;          % PeerInfo
PKp_e           = PeerId -> PK_t;            % Public key (peer)
Np_e            = PeerId -> N_t;             % Peer nonce
Ns_e            = PeerId -> N_t;             % Server nonce
Noob_e          = PeerId -> Noob_t;          % Noob nonce
OOB_e           = PeerId -> Nat;             % OOB retries
Type_e          = PeerId -> Type_t;          % Expected type
Kms_e           = PeerId -> K_t;             % ECDH key (server)
Kmp_e           = PeerId -> K_t;             % ECDH key (peer)
```

Listing 4.6: Database entries

The following actions are supported through the interface:

- Query and update:
  - all data stored for a given *PeerId*
  - the expected type of the next message
  - the number of failed OOB messages received
- Update only:
  - the nonce value of the peer
  - the derived keys for both the server and the peer
- Reset the database entry of a given *PeerId*

Although the database is a separate process from the server, communication between the database and the server is considered to be reliable and the transactions atomic. Each visible action consists of a pair of actions (* and *_DB) synchronised to form an atomic multi-action (_MA).

4.2 Time and Timeouts

Modelling time and timing properties is a central part of verification of communication protocols. The unreliability of the communication channels requires protocols to consider messages delivered with a delay or even disappearing completely. Hence, the ability to model time is generally a vital

feature of any modelling language. There are two alternative ways of dealing with time-dependent processes in a nondeterministic model of a system. The first, is expressing it as a *timed transition system* (TTS).

A timed transition system TTS can be defined as follows [23]:

$$TTS = (S, ACT, \rightarrow, \rightsquigarrow, s, T) \tag{4.1}$$

where S is a set of states, ACT is a set of actions, $\rightarrow \subseteq S \times Act \times \mathbb{R}^{>0} \times S$ is a transition relation, $\rightsquigarrow \subseteq S \times \mathbb{R}^{>0}$ is the idle relation, $s \in S$ is the initial state, and $T \subseteq S$ is the set of terminal states.

For example, consider the transition system depicted in figure 4.4. Similar to the LTS in figure 2.3, the action A transitions the system from the initial state s_0 to the terminal state s_1. However, in this case, the transition is only allowed to happen within the first 9 time units. After that, the process timeouts and ends up in the deadlock state s_2.

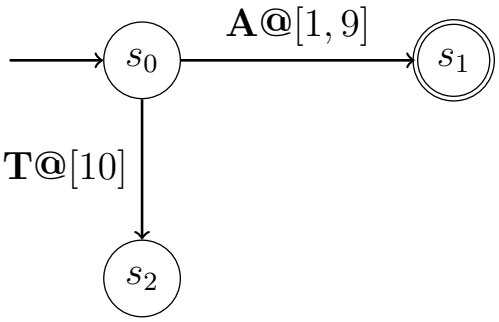

Figure 4.4: Timed Transition System of a Process

Another way of modelling time and timeouts is to state that a timeout *can* possibly happen, split the execution path in two distinctive futures, and explore both. The downside of this technique is the number of states and transitions it creates. For every nondeterministic choice, two paths will be created, which makes the state space of the model grow significantly. On the other hand, disabling some choices allows to adjust the state space to correspond to the attributes under test.

To simplify the model and avoid explicit definition of time for each process, we modelled the protocol following the second alternative. For example, as shown in listing 4.7, the server *can* timeout whenever it is idle in the Registered (4) state. If a test case is not dependent on reaching the Reconnecting (3) state, the timeout action can be blocked, which prevents it during the execution.

```
% Allow Mobility/Timeout/Failure to happen for the server
+ (state(PeerId) == s4 && type(PeerId) == no_type) -> (
    SERV_STATE(PeerId, s4)
    . Database()
    + MOBILITY_TIMEOUT_FAILURE
    . SERV_STATE(PeerId, s3)
    . Database(
        state = state[PeerId->s3]
    )
)
```

Listing 4.7: Server timeout

Chapter 5

Verifying Protocol Properties

In this chapter, we describe how the protocol properties were verified and which changes were made in the specification and its implementation. We begin by discussing the general procedure of protocol verification in our model and describe how the different types of attributes were tested. After that, we present the significant improvements and clarifications made to the specification based on our modelling and verification results. Finally, we discuss a major change in the implementation to fix errors caused by misinterpretation of the specification.

The implemented verification queries can be divided into four categories: (1) queries for the reachability of normal states, (2) liveness and deadlock queries, (3) error state queries, and (4) error recovery queries. The first group consists of tests for verification of normal protocol behaviour, i.e. tests that check whether each exchange can be completed with the expected results. Successful completion of these queries denotes that all initiated peers were able to connect to servers and that each of them performed a successful Reconnect Exchange. Furthermore, the liveness queries ensure that the state machines cannot end up in an unrecoverable combination of states. The additional queries are concerned with reachability of error states and recovery from them. First, they check whether or not any of the error states are reachable during normal protocol execution or after an error, such as message corruption, is caused intentionally. If an error state can be reached, a follow-up query determines if it is always possible to recover from it and end up back in the Registered (4) state. Some errors, such as mismatching version numbers, are always impossible to recover from, whereas others can be expected to occur frequently are are recoverable.

Auxiliary Actions

In our model, verification of reachability properties is based on the detection of informational transition events, introduced exclusively for the purpose of model checking. The property language of mCRL2 (modal μ-calculus) is an event-based query language, i.e. rather than relying on detection of states, it detects the transitions between them. As a result, instead of directly verifying the reachability of state combinations, we were forced to introduce some auxiliary actions to inform the queries of the current states of peers and servers. An auxiliary action resembles any other action in the model, with the exception that it is independent (i.e. not part of any multi-action) and without side effects. It can be triggered before or after an event takes place and queried by the μ-calculus formulas. This way, we are able to not only query actual transitions but also the status of any state machine in the model. Similarly, errors and other anomalies can be detected and queried by creating events that are triggered in the error state.

The auxiliary actions of the model include:

Action	Event
SERV_STATE(PeerId, State)	a server enters a new state for some *PeerId*
PEER_STATE(PeerId, State)	a peer enters a new state
LOG_ERROR(ErrorCode)	the server sends or receives an error message
MESSAGE_LOST(Type)	a message is lost in a channel
MAX_PEERIDS_REACHED()	no more *PeerIds* can be generated
MAX_NOOBS_REACHED()	no more *Noob* values can be generated
MAX_NONCES_REACHED()	no more nonces can be generated

5.1 Simulating Expected Protocol Behaviour

Our main approach to protocol verification is based on simulation of expected behaviour. As described in chapter 3, both the server and the peer maintain their separate state machines indicating the status of the connection. Thus, the execution of the protocol can be described as pairs of server-peer states, e.g. server in state 2, peer in state 1. We can determine whether the expected combinations are reachable and whether the unexpected combinations are unreachable by exhaustive exploration of the state space. In case the model reaches an undesired state combination, it can be backtracked towards the initial state until the origin of the issue is found.

Much like in traditional software testing, we created a collection of tests for all (5×5) 25 state combinations (0-4 for both parties) and 11 error

states (see section 5.3). Each query examines a pair of server-peer states and returns *true* if there exists a path leading from the initial state to the given combination of states, and *false* otherwise. For example, the reachability query in listing 5.1 states that for each *PeerId* n, both the server and the peer can enter the Registered (4) state in a series of transitions. In case the query returns false, one of the endpoints might have reached a deadlock (or livelock) prohibiting it from ever completing the Completion Exchange.

Since µ-calculus is an event-based language, testing reachability requires querying the transitions leading to the combination of states under test, rather than querying the actual states. For example, to test whether the state combination (4,4) is reachable, we use the following formula (E = exists, F = finally):

$\mathrm{EF}(S_4, P_4)$	(state-based temporal logic)
$\mathrm{EF}(S_4 \wedge \mathrm{X}(\neg S_x \mathrm{U} P_4))$	(event-based temporal logic)
$\mu Y.((S_4, P_4) \vee \Diamond_t Y)$	(µ-calculus)

To determine whether a state combination is reachable, it is sufficient to find *one* path leading to it. The aforementioned query can be simplified to a single case, in which the server enters state four, immediately followed by the peer entering state four (see listing 5.1). Successful completion of this query means that *at least one* path exists that leads from the initial state to the state combination (4,4). In other words, it shows that the combination is reachable, but does not guarantee that it can always be reached.

```
% Eventually, the server enters state 4, immediately
% followed by the peer entering state 4
forall n:Pos . val(n <= 1)
  => <true* . SERV_STATE(n, s4) . PEER_STATE(n, s4)> true
```

Listing 5.1: Reachability query for states (4,4)

Even though simplification to a single case works for reachability queries, it is not sufficient to show that a state combination is *unreachable*. Even if one path is shown to not exist in the state space, there might be others that eventually lead to the same result. For this reason, unreachability tests are slightly more complicated. Instead of testing one case, we include all possible paths, in which one of the parties transitions first and the other one eventually follows. Listing 5.2 shows how the (unreachable) state combination (4,1) is tested.

```
% After the server enters state 4, the peer will not be able
% to enter state 1 until the server transitions again.
forall n:Pos . val(n <= 1)
  => [true* . SERV_STATE(n,s4)
      . !(SERV_STATE(n,s0) || SERV_STATE(n,s1) ||
          SERV_STATE(n,s2) || SERV_STATE(n,s3))*
      . PEER_STATE(n,s1)] false

&&

% After the peer enters state 1, the server will not be able
% to enter state 4 until the peer transitions again.
forall n:Pos . val(n <= 1)
  => [true* . PEER_STATE(n,s1)
      . !(PEER_STATE(n,s0) || PEER_STATE(n,s2) ||
          PEER_STATE(n,s3) || PEER_STATE(n,s4))*
      . SERV_STATE(n,s4)] false
```

Listing 5.2: Reachability query for states (4,1)

All reachable state combinations (excluding temporary states after a user reset or an error) are listed in table 5.1. As described in section 3.2, reaching the OOB Received (2) state is a special case that requires the negotiated OOB direction to allow the other party to send OOB messages. In case the direction is set to 2 (server-to-peer), only the peer can enter state 2 and, correspondingly, if the direction is 1 (peer-to-server), only the server can enter state 2. The third case, in which the direction is set to 3, allows either one of the endpoints to send an OOB message and can in some rare situations lead to both parties to enter state 2 simultaneously.

Results: table 5.1 shows the results of the reachability queries. As expected, all good states are reachable and most bad states are unreachable. However, there are several unwanted states, all of which have one of the parties in state four and the other one in state one or two. There are two possible explanations for this: (1) there is an issue in the protocol that in some situation causes an unrecoverable error after the Completion Exchange, or (2) the queries fail because the transition events are asynchronous. In fact, the failure may be caused by either reason. Unfortunately, the limitations of the query language means that we cannot properly test the transition to the persistent state. The protocol issue is currently verified, but not fixed. It is further explained and investigated in section 5.4.

Server state	Peer state
1	1 2* **4***
2	1** 2*** **4****
3	3 4
4	**1**** **2*** 3 4

* = OOB direction 2 or 3, ** = OOB direction 1 or 3, *** = OOB direction 3

Table 5.1: Reachable state combinations

5.2 Deadlocks and Liveness Properties

One of the main goals of our model is to verify the absence of persistent error states in the EAP-NOOB protocol. We decided to model the protocol with the formal verification language mCRL2, because it is designed for validation and verification of distributed systems [23] and suitable for examination of liveness and deadlock properties. A deadlock occurs when the system enters a state in which no more actions can be performed and the system cannot progress any further by transitioning to another state [35]. A livelock, on the other hand, allows the system to perform some actions but prevents it from making significant progress in the protocol. It essentially creates a loop-like situation, in which the process seemingly performs work, but never actually advances as intended. As shown in section 2.1.1, detecting a deadlock is generally a straightforward task. Unfortunately, detecting a livelock is slightly more complicated and requires consideration of time-related properties regarding the progression.

Consider the following formula (G = globally):

AG EF(S_4, P_4)	(state-based temporal logic)
AG EF$(S_4 \wedge \mathrm{X}(\neg S_x \mathrm{U} P_4))$	(event-based temporal logic)
$\nu Y.(\mu Z.((S_4, P_4) \vee \Diamond_t Z) \wedge \Box_t Y)$	(µ-calculus)

The formula states that there **always exists** a path that **finally** ends up in state (4,4). In other words, the protocol is always capable of returning back to the persistent state, regardless of the current state. Listing 5.3 shows the mcf formula for this property. However, because only a limited number of nonces that can be generated in the model, this query will not terminate successfully. Furthermore, even if the number of nonces was taken into account in the query, the possibility of persistent failure due to loss of last message (see section 5.4) would cause the query to fail.

```
% It is always possible to return to state (4,4)
[true*]<true*>(S4,P4)
```

Listing 5.3: Liveness query

Results: no deadlocks can occur in the protocol, unless it is under a persistent denial-of-service attack. We did not manage to verify the absence of livelocks.

5.3 Error States and Error Recovery

An important part of the verification procedure was to examine the behaviour of the protocol when introduced to issues that resulted in one or both of the parties to end up in an error state. These states are not a part of the normal protocol behaviour and should therefore be examined in detail in order to verify correct behaviour and to avoid deadlocks. For each error that can occur, the protocol needs to define a recovery procedure, as an attacker might otherwise be able to cause a persistent failure by forcing one of the parties to enter an unrecoverable error state.

In the EAP-NOOB protocol, errors are handled by sending the other endpoint a special error-notification message (type 0) and, if possible, recovering to a persistent state (see section 3.2). The protocol specification contains a list of possible errors and defines the correct course of actions for each exchange that should be taken after sending or receiving an error message. In addition to a unique code, each error has a description to inform the user of what the cause of the issue was. All currently implemented and tested errors in the model are listed in table 5.2. Some additional errors are defined by the specification but it is not possible to test them with mCRL2 and thus they are excluded from the model.

Error Code	Error Message	Recoverable?
1003	Invalid data	Yes
1004	Unexpected message type	Yes
1005	Unexpected peer identifier	Yes
1006	Unrecognised OOB message identifier	Yes
2002	State mismatch	Yes*
3001	No mutually supported protocol version	No
3002	No mutually supported cryptosuite	No
3003	No mutually supported OOB direction	No
4001	MAC verification failure	Yes
5002	Invalid server info	No
5004	Invalid peer info	No

* = user action required

Table 5.2: Error Codes and Messages

The recovery procedure after either one of the endpoints reaches an error state is a specific case of liveness that requires its own set of queries to be tested. Error and error recovery queries check whether a given error can occur, and if it can, is it always possible to recover from it. For example, the detection query for error 3002 in listing 5.4 declares that there does **not exists** a path leading from any state to a state in which the error occurs.

```
% Error 3002 never occurs
[true* . LOG_ERROR(E3002)] false
```

Listing 5.4: Error query for error 3002

Most errors are not supposed to happen during normal protocol execution. However, in some cases, certain errors are allowed to occur during an exchange and should not be considered abnormalities, as long as the protocol can recover from them. In these cases, after the error occurs, the model checks whether it is always possible to recover from the error and only reports it if there is an execution path in which the system cannot recover. For example, the recovery query for error 1006 in listing 5.5 declares that there **always exists** a path leading from the error state to the Registered (4) state, as long as a new nonce can still be generated. However, this is only true for some of the possible error states and, hence, errors can be divided into two categories: (1) recoverable errors, and (2) unrecoverable errors. Unrecoverable errors, such as mismatching version numbers, are persistent and

need to be manually solved by a user.

```
% If it is still possible to generate a new Noob, it is
% always possible to recover from error 1006
forall n : Pos . val (n <= 1)
  => [!MAX_NOOBS_REACHED* . LOG_ERROR(E1006)]
       <true* . SERV_STATE(n, s4). PEER_STATE(n, s4)> true
```

Listing 5.5: Recovery query for error 1006

Another case in which errors are expected is when issues are purposefully introduced in the model in order to examine how it behaves when something goes wrong. One way of doing this is to compromise one or more of the in-band channels. Since most of the communication is done over these channels, values can be modified and dropped as needed. For example, changing the identifier *NoobId* in the Completion Exchange should lead to error 1006 (unrecognised OOB message identifier) since the receiver does not recognise the nonce. To enable message spoofing and modification, the in-band channel can be initialised as *compromised*, as shown in listing 5.6.

```
% In-band channel: Server to Peer
%   compromised : true, if the channel is controlled by an attacker,
%                 false otherwise
ServerToPeerChannel(compromised:Bool) =
  ...
  % Modify values if the channel is compromised
  (compromised) -> (
      (type == t4) -> (
          % Spoof NoobId
          RECV_EAP_REQ_O(type, PeerId, req4(NoobId(data)+10, MACs(data)))
          . ServerToPeerChannel(false)
      )
  ...
;
```

Listing 5.6: In-band server-to-peer channel

To avoid a state explosion, the number of unique nonces and identifiers that can be generated is limited to a minimum, which in some cases means that the system enters a deadlock when it tries recover from an error. These deadlocks are ignored when testing error recovery, because they would not occur in an implementation which allows generating an unlimited number nonces and identifiers. In the model checking process, this is done by limiting the queries to cases where the maximum number of generated values has not yet been reached. Furthermore, the attacker is limited to one spoofed message per channel (as shown in listing 5.6). If the attacker were allowed to

continue spoofing messages, the protocol would end up in a persistent livelock and would never be able to successfully finish a Completion Exchange.

Results: no unrecoverable errors can occur and the protocol is capable to recover from non-persistent denial-of-service attacks and modified values.

5.4 Persistent Failure Due to Loss of Last Message

Since EAP often works on top of an unreliable lower layer [45], packets may end up disappearing anywhere between the sender and the receiver, often without either one realising it. Although many attempts have been made to create reliable communication protocols, message loss is an inevitable consequence of communication over lossy channels and should therefore be expected to occur at any given time. The reasons for it can be divided into two categories: intentional and unintentional causes. Possible reasons for dropped packets include radio link failures, congested channels and unreachable nodes in the network. Furthermore, by placing itself between the two endpoints, an attacker can exploit these possibilities to purposefully cause persistent failure and denial-of-service. For these reasons, the ability to drop packets was added to the in-band channels in the model.

In the context of distributed systems, the *state of knowledge* refers to a set of local or global information. Each processor makes its decisions based on its local state of knowledge, whereas a distributed system of processors requires groups of local knowledge states from all participants in order to make decisions [25]. Communication in a distributed system can be viewed as advancing in the knowledge hierarchy, ranging from *distributed knowledge* to *common knowledge*. Distributed knowledge includes information issued among the members of a group without a guarantee of it being delivered, whereas common knowledge is "public" information, guaranteed to be known by all parties. However, as shown by Halpern and Moses [25], common knowledge in a strict sense is impossible to achieve in a distributed system where messages may be dropped.

Many protocols, such as TCP [28], rely on acknowledgement (ACK) packets to inform the other endpoint that the previously sent message was delivered correctly. However, asynchronous communication protocols all suffer from a common issue: how can the delivery of the last message in a sequence of packets be guaranteed? The reason for why this is impossible can be intuitively understood by considering a system with two endpoints (see

figure 5.1). Each party can only perform an action iff the counterpart is guaranteed to perform the same action simultaneously. Each ACK confirms the delivery of the previous packet, but requires in turn a new ACK to be acknowledged. No matter how long this goes on, the last message remains unacknowledged and can be dropped without either of the endpoints realising it.

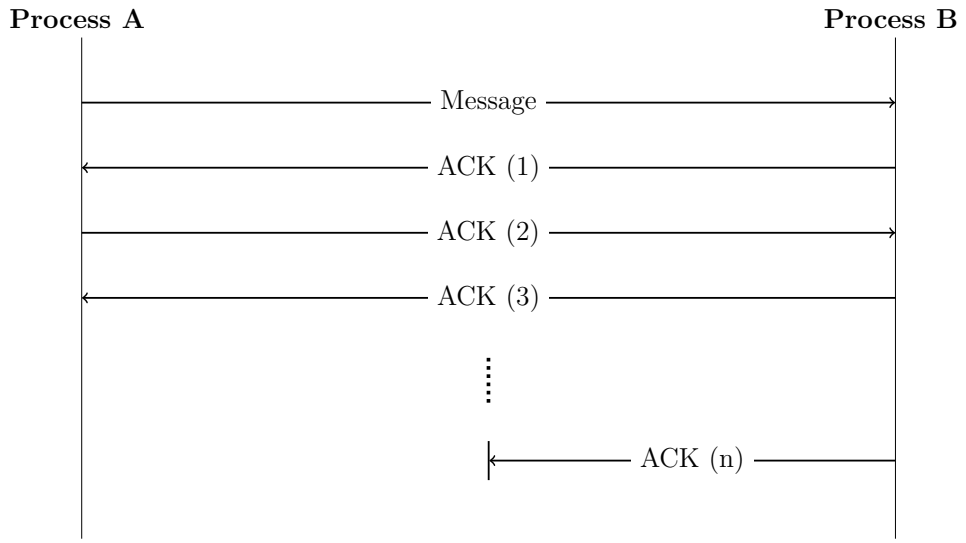

Figure 5.1: Achieving Common Knowledge in a Distributed System

As described in section 3.2, each exchange in the EAP-NOOB protocol ends with either an EAP-Success or an EAP-Failure message sent to the peer. For the server, sending this message implies that the exchange was successfully completed and it can transition to the next state. Correspondingly, the peer can either consider sending the last response, or receiving the success or failure message as the end of a successful exchange. In most cases, this works as intended and ensures that both parties are aware of each other's state. However, if the last message is lost, one of the parties will transition to the next state before completing the exchange. Since a state mismatch is considered to be an unrecoverable error, this causes a persistent state of failure if it happens during the Completion Exchange or when updating values needed for key-derivation. Implicit success indication can be introduced to avoid issues caused by lost success or failure messages, but similar to the case depcited in figure 5.1, this only shifts the process of acknowledgements back by one step.

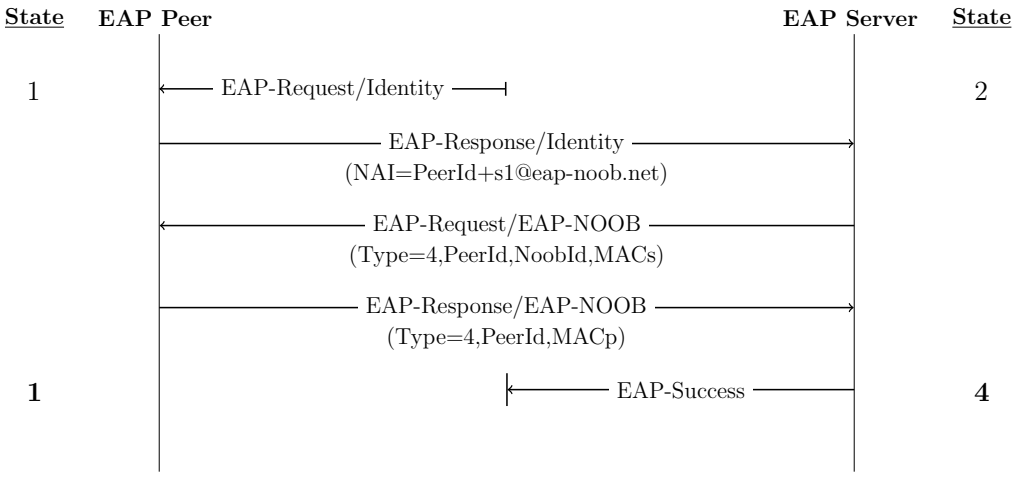

Figure 5.2: Loss of the Last Message in EAP-NOOB

Packet loss between a peer and a server can be modelled as a special case of liveness. For example, consider the Completion Exchange depicted in figure 5.2. It starts with the peer in the Waiting for OOB (1) state and the server in the OOB Received (2) state. The exchange begins with the peer receiving an identity request from the authenticator and ends with the server sending an EAP-Success message, which in this case is lost in the channel. As a result, the peer remains in state 1 waiting for the missing packet, whereas the server transitions to the Registered (4) state. Now, if the peer tries to continue the exchange or start over, the server will consider it a state mismatch, send an error message and inform the user that a reset is needed to recover. The protocol specification does currently not define how this could be avoided but acknowledges the issue.

Results: failure to deal with the loss of the last packet confirmed in the verification.

5.5 Recovering From A Rejected NoobId

Generating (pseudo-)random nonces is part of the EAP-NOOB exchange process for both servers and peers. During the Initial Exchange (and possible Reconnect Exchanges), both the peer and the server generate their own connection-specific nonces (Ns and Np) for MAC calculation and key derivation. Furthermore, prior to the Completion Exchange, one or both of the parties generate a nonce *Noob* and send it over the OOB channel together with the peer's identifier *PeerId* and a cryptographgic fingerprint *Hoob*. The

receiver of the message calculates a *Hoob* with the corresponding values and accepts the nonce if it is still valid and if the fingerprints match. The validity period of the nonce depends on an application-specific timeout value, *NoobTimeout*, which determines how long after its generation a nonce is still usable.

After receiving a valid OOB message, the receiver transitions to the OOB Received (2) state to indicate its readiness for the Completion Exchange. In normal cases, the exchange begins shortly after this, and the received *Noob* value remains valid. However, since each nonce is given an expiration time when created, it can expire before the Completion Exchange, which causes the sender to forget it. The receiver, on the other hand, may still accept the OOB message and attempt the Completion Exchange with the expired nonce. The *NoobId* calculated from the nonce will now be rejected in the Completion Exchange and an error message (error code 1006) will be sent to the other party indicating that the nonce was invalid. In version 02 of the EAP-NOOB draft [2], if the invalid *Noob* is the only received nonce, this will lead to a deadlock situation, as no new OOB messages are accepted in state 2.

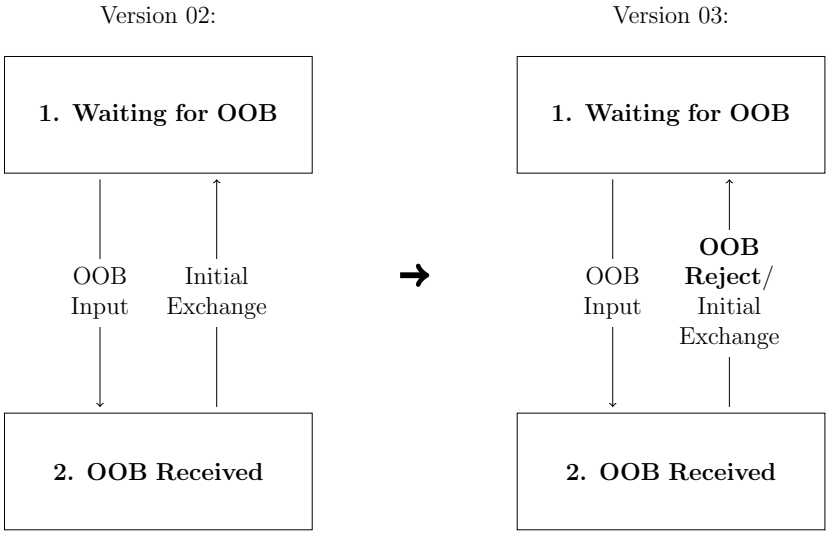

Figure 5.3: Updated Completion Exchange

Fortunately, ending up in a deadlock because of an expired nonce is easily avoidable by adding an additional transition from state 2 to state 1 (see figure 5.3). If the Completion Exchange fails due to an expired or invalid *Noob*, both parties revert back to the Waiting for OOB (1) state and start

over. The additional transition was included in version 03 of the draft [3] and verified by modelling an execution path in which the received nonce is expired, which causes error 1006 to occur (see section 4.2 for further details about modelling timeouts).

In addition, a clarification was made regarding any additional OOB messages received after accepting a nonce and transitioning to the OOB Received (2) state. When receiving an OOB message in the Waiting for OOB (1) state, the receiver has two options: (a) if the nonce *Noob* has not yet expired and the locally calculated fingerprint *Hoob* matches the one in the message, transition to state 2, or (b) reject the message and notify the user. After accepting a valid OOB message and moving to state 2, no more OOB messages should be accepted. However, these messages may be buffered for later use in case an expired nonce or other failure causes the party to revert back to the Waiting for OOB (1) state.

Results: improvements and clarifications to the specification to avoid a deadlock situation caused by an expired nonce.

5.6 Rejecting Unexpected Messages

Throughout the modelling process, we discovered various ambiguities in the specification, some of which caused the model to contain some incorrectly implemented parts. Each detected inconsistency was subsequently compared against the corresponding part of the EAP-NOOB software implementation. This way, we managed to improve the validity of the model, as well as that of the implementation.

One of the biggest flaws that we detected concerns how to deal with unexpected messages. This issue was discovered when validating the model and shown to exists in the implementation as well. In order to avoid unwanted messages, both the server and the peer maintained information about the ongoing exchange in a variable and only accepted messages of types that are part of that exchange. For example, after initiating the Initial Exchange, the message types accepted by the peer included EAP-Requests of type 1 and 2, and the EAP-Failure that ends the exchange. This works well in most cases because the exchange process (mostly) follows a linear request-response pattern and any lost messages are retransmitted by the underlying EAP layer. However, in the case where the negotiated OOB direction is set to 3 (both S2P and P2S are acceptable directions), this causes issues in a system with multiple parallel channels that messages can be sent over.

Consider the following situation:

1. The OOB direction is negotiated to 3, meaning that both the server and the peer are allowed to send OOB messages.
2. Both parties are in the Waiting for OOB (1) state, generate *Noob* nonces and send them to the other endpoint in OOB messages.
3. Due to a slow S2P OOB channel, the P2S message gets delivered first and the server transitions to the OOB Received (2) state.
4. The peer initiates a Completion Exchange, but before the exchange can be completed, one of the messages gets temporarily stuck in the in-band channel.
5. Before re-initiating the exchange, the S2P OOB message is delivered and the peer transitions to state 2.
6. Now both parties are in state 2, which is treated similarly to the case in which the server is still in state 1.
7. A new Completion Exchange is initiated by the peer. Before it finishes, the message held up in the channel is finally delivered causing two simultaneous Completion Exchanges to run in parallel.
8. Eventually, one of them finishes, which allows both parties to transition to the Registered (4) state.
9. The registration is now complete, but might result in a MAC verification failure in case of a Reconnect Exchange, as the derived keys do not necessarily match if different (*Noob*) nonces were used during the key derivation.

The problem was solved by replacing the variable that keeps track of the ongoing exchange with a variable that stores the type of the previously sent message. Instead of receiving any type of message belonging to the ongoing exchange, the server only accepts a response corresponding to the previously sent request. For example, after sending an EAP-Request of type 1, the server expects to get an EAP-Response of type 1 in return. Any other messages are discarded and an error message (error code 1004) is sent to notify the other party that a message was rejected.

Results: major improvements to the EAP-NOOB software implementation.

Chapter 6

Testing the Cryptographic Implementation

Our goal for this thesis was to implement a symbolic model of the EAP-NOOB protocol with the formal specification language mCRL2. This model was mainly used for verification of properties in the current protocol specification, as well as verification of the improved version for the upcoming draft. Throughout the model-checking process, we discovered various inconsistencies, ambiguities and other minor defects in both the specification and the implementation, all of which were subsequently corrected or clarified.

However, as mCRL2 is mainly intended for simulation and state space exploration, our model is not capable of verifying cryptographic output. Instead of performing actual calculations on cryptographic primitives, it simply represents the results as comparable structures that contain the input values. This is sufficient to check the correctness of the protocol on an abstract level, but it cannot detect errors in the the cryptographic implementation. Errors can be caused, for example, by ambiguities in message formats and incompatibilities between cryptographic libraries. Additional verification approaches are required in order to ensure that the cryptographic aspects of the protocol are sufficiently described in the specification and work as intended in the implementation.

In addition to verifying the protocol specification with model checking techniques, we created a test script[1] for message generation based on test vectors and previously generated nonces. It is, however, not a complete implementation of the protocol, but rather a complementary tool intended for verification of the outputs from actual implementations. The script was implemented with Python and the cryptographic calculations are mainly based

[1] https://github.com/tuomaura/eap-noob/tree/master/test-vectors

on two modules: the *cryptography.hazmat* module for low-level cryptographic calculations[2] and the *PyNaCl* module[3] for (public and private) key generation and calculation. To ensure that the script produces correct outputs, we used the test vectors [36] provided for the X25519 function for Elliptic Curve Diffie-Hellman calculations and compared our results with the expected output.

Since the script is not intended for simulation of the protocol, it needs to be provided with the device-specific values, generated nonces and ECDH keys as input. These values are then used for the cryptographic calculations (hashes, MACs and key derivation) and printed out in a readable format for easy comparison with the output of the actual implementation. Each message type, excluding failure and success messages, is included in the output, and everything is printed out as it would be received through the channels, making it a mixture of plaintext and base64url [30] encoded values.

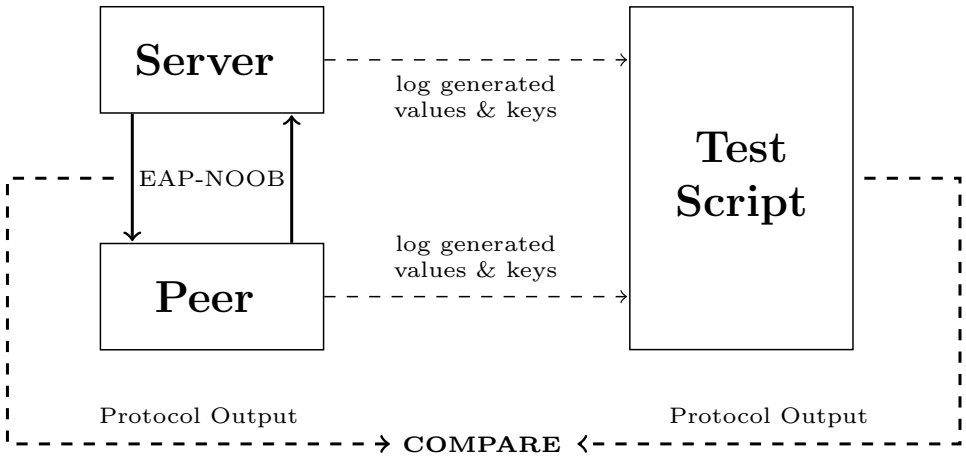

Figure 6.1: Comparison of Cryptographic Output

Figure 6.1 outlines the intended verification procedure of a protocol execution. The implementation, depicted on the left side of the figure, performs all the usual steps of connecting a device to a server and stores both the generated values and the protocol output in message logs. After the last exchange is completed, the script is configured with the generated values and a corresponding output is produced. Any differences in the two outputs should be considered as possible errors and further investigated. Listing 6.1 shows the output of a Completion Exchange, which contains the 16-byte identifier *NoobId* computed with a one-way function of the generated *Noob* nonce, as

[2]https://cryptography.io
[3]https://pynacl.readthedocs.io

well as the MACs calculated witht the HMAC [34] function for both the server (MACs) and the peer (MACp).

```
====== Completion Exchange ======

Identity response:
   u9N3BQhuKmDFQ0n5y94nvt+s2@noob.example.com

EAP request (type 8):
   {"Type":8,"PeerId":"u9N3BQhuKmDFQ0n5y94nvt"}

EAP response (type 8):
   {"Type":8,"PeerId":"u9N3BQhuKmDFQ0n5y94nvt","NoobId":"
   LBtGFeOkMeg4nA2oAlolmA"}

EAP request (type 4):
   {"Type":4,"PeerId":"u9N3BQhuKmDFQ0n5y94nvt","NoobId":"
   LBtGFeOkMeg4nA2oAlolmA","MACs":"SFGm-
   HaUmxSDMxA_95nVo8AQ_Jmr6ZjraltxmCIgC4Q"}

EAP response (type 4):
   {"Type":4,"PeerId":"u9N3BQhuKmDFQ0n5y94nvt","MACp":"
   Pr6sCG_b3d8T0NQp4_M9dB40-Gg5Hb-_O89PYfxx0X4"}
```

Listing 6.1: Output of the Completion Exchange

Results: Clarifications introduced in version 03 of the protocol draft [3]:

- In addition to specifying an ECDH curve and a hash function, each cryptosuite must define how the public key should be encoded as a Jason Web Key (JWK) [29] object, since differing encodings will lead to mismatching keys. Furthermore, the naming conventions and ordering of the parameters has to be mutual. This clarification was made to ensure consistency and avoid ambiguous interpretations of the key formats mentioned in the specification.

- EAP-NOOB requires some parameters to be base64url [30] encoded and offers it as an option for others. Each base64 encoded character represent six bits of data. The encoding process splits the input into groups of three 8-bit binary values combined into a 24-bit sequence,

which is encoded as four base64-encoded characters. If the output is not a multiple of four, '=' is used for padding. In the case of EAP-NOOB, padding does not make a difference as the values are of fixed length. However, if one party decides to add padding to the encoded values and the other one decides not to, it might cause problems with decoding and MAC calculations. Thus, in version 03 of the draft, a clarification was made to indicate that no padding should be added by either party.

- Although nonces may be sent over the channel as base64-encoded values, they should be decoded to raw bytes for the key derivation function. This was already implied in the previous draft, but a clarification was made to emphasise it further, as in other cases message fields are expected to be copied verbatim without any modifications.

- The explanations of *PeerId*, *Realm* and *NAI* were simplified and an appendix suggesting values for *ServerInfo* and *PeerInfo* was added. Although the content remained unchanged, these clarifications were made to improve the readability and unambiguity of the specification.

Chapter 7

Discussion

In this chapter, we discuss the modelling process, the contributions we made and reflect on future work that still needs to be done. We begin with summarising the most important outcomes from the project and evaluate the overall implementation process. We finish the chapter by suggesting additional measures to be taken in order to improve the reliability of the protocol before standardising it.

7.1 Contributions

In this thesis, we created a symbolic model of the EAP-NOOB protocol with the formal specification language mCRL2. We modelled a system with multiple servers, peers and channels, and verified that each individual device managed to authenticate to a server as specified in the protocol. Furthermore, we formalised a set of properties that describe reachability of states, lack of errors and recoverability from error states. To examine how the protocol handles unexpected behaviour, we intentionally introduced issues by compromising one or more of the in-band channels and purposefully manipulated data. Finally, we performed multiple simulations and backtracked traces from reachable error states to figure out how to patch the protocol against the issues we found. We also implemented a test script for cryptographic verification that could not be done with our model due to its design and implementation language.

The main contribution of this thesis was to provide feedback regarding issues found during the modelling and verification processes (see chapters 5 and 6) for the development of the next IETF draft, as well as to verify the reliability of the updated protocol version. We managed to create a complete model of the protocol, including a set of automated tests for property

verification. The final version of the model, as well as the test script, were published together with the implementation[1] and are available for supporting future development of the protocol.

One of the main results from our work was to discover unrecoverable error states due to lost messages (see section 5.4), which can be exploited in order to cause persistent failure in the system. Forcing IoT-devices to enter deadlock states may cause significant financial damage if the number of devices is large. Although we only examined one protocol, it is reasonable to assume that other similar protocols might be affected, which requires future work to be clarified.

7.2 Reflections

The goal of our work was to improve the reliability of the EAP-NOOB protocol and its implementation with formal verification techniques. We created a symbolic model of the protocol with the mCRL2 verification language and performed exhaustive state space exploration to determine its behaviour in various situations. Our model detects informational actions that describe the occurrence of events of interest, which allows us to check the model for liveness properties, deadlocks, reachability of error states and error recovery. We managed to simulate the protocol behaviour when exposed to not only normal but also abnormal situations, such as dropped or modified messages. Furthermore, our test script (see chapter 6) verifies the cryptographic calculations made in the protocol implementation and presents the overall structure of the exchanges.

Throughout the modelling process, we encountered various minor problems regarding how to model a specific feature or functionality. Most of the issues were caused by either limitations in the verification language or ambiguities in the specification (see section 5.6). The language limitations varied from minor inconveniences, such as limitations in the implemented data structures, to severe restrictions that affected our design decisions. Issues caused by misinterpretations of the specification were relatively easy to fix after discovery, whereas a language limitation forced us to find an alternative way to solve the problems it affected. The language limitations we discovered include:

- **Declaring aliases for data types.** In an attempt to unify the protocol specification and model as much as possible, we applied the type alias declaration feature of mCRL2 to our model. This feature allowed

[1] https://github.com/tuomaura/eap-noob/

it to contain custom data types that were mapped to existing ones during the compilation process. For example, a *PeerId* could be declared with a type called *PeerId_t*, even though it is was represented as a natural number during the calculations. However, this feature was incorrectly implemented in the verification language, which caused exponential compilation times when applied to the model (see appendix A for more details). Although not a crucial part of the modelling process, finding the cause of the issues was time consuming.

- **Preprocessing the specification file.** One of our main design goals for the model was to make it as easily adaptable to different scenarios as possible. Ideally, the user should be able to use the same model for all verification tasks and simply declare the settings for the protocol (i.e. determine the number of servers, peers, channels, and set the limitations for generated values) prior to executing a collection automated tests. However, as the entire specification is located in a single file, the only way to change parameters is to manually edit that file. Furthermore, testing the model is based on queries and each individual query has to be executed separately. In the end, we managed to automate the compilation process, execution of the tests and some other minor tasks related to simulation and visualisation of the model. This was easily done with a makefile and some shell scripts, but the only way to change the parameters is still to edit the specification file. Unfortunately, there does not seem to exist a sensible way of preprocessing the specification file, other than to write a separate program that modifies it line by line.

- **Creating and running parallel processes.** A crucial part of verifying the protocol involves modelling a typical client-server setting, in which one server communicates with multiple clients (peers) simultaneously. One way to implement this is to have a master process (server) that creates new threads as required. However, mCRL2 only supports sequential processing and does not allow a process to create an unspecified number of new processes during the model checking phase. Because of that, we had to settle for simulating the behaviour with a limited number of parallel processes for the peers and a single-threaded server process that sequentially processes client requests.

Although we managed to overcome all the issues mentioned in this section, one way or another, there is still room for improvement and possibly better ways to solve them. For instance, instead of settling for sequential behaviour for the server, it might be possible to create a thread pool of pre-instantiated

threads for new connections. This way, when a peer connects to the server, one of the threads can be assigned to it and the server can continue waiting for new connections. Once the thread is done working, it simply returns to the thread pool to be used later. Deciding the amount of threads before compilation would overcome the inability to create new processes during the model checking and would simulate a real server with limited resources.

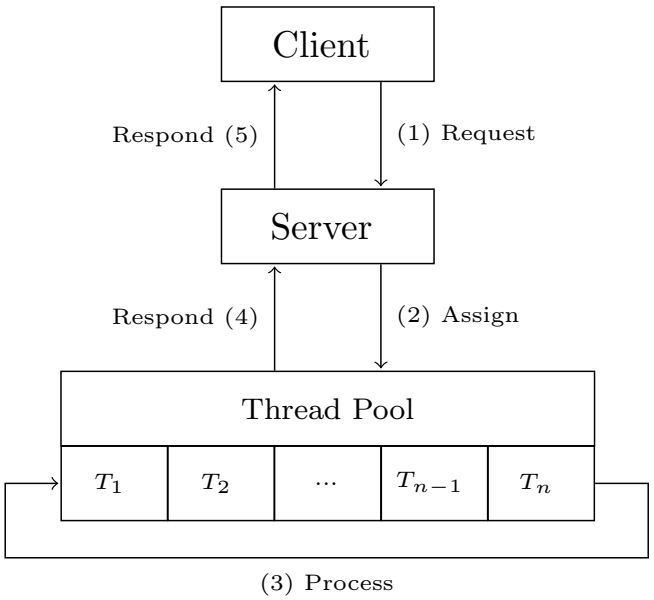

Figure 7.1: Conceptual Overview of a Thread Pool

Furthermore, even though we limited the state space as much as possible, most values are still explicitly initiated as numbers or lists. Instead, we could reduce the state space even further with symbolic modelling techniques that cannot be done with the mCRL2 language. The size of the state space for each possible direction (with one server, one peer and the minimum possible number of nonces and identifiers generated) is listed in table 7.1. The reason for the large state space in the third row is that OOB direction 3 includes both the previous cases as well as the special case of both endpoints sending an OOB message. In addition, the maximum number of generated nonces has to be increased in order to avoid problems with timeouts when examining the special case.

Direction	States	Transitions
1	2 372	7 777
2	2 576	8 344
3	362 277	93 854

Table 7.1: State Space of the Model

7.3 Future Work

The model we created for this thesis was developed with the mCRL2 language with the purpose of verification of reachability properties by exhaustive state space exploration. Consequently, all properties were tested with auxiliary event actions, as described in chapter 4. Although convenient for verification of properties related to state reachability and error recovery, this approach does not support modelling basic security considerations, such as secrecy or authentication. Each sender is blindly trusted and messages are accepted as a valid part of the ongoing exchange, as long as their content and format resembles what is actually expected. Fortunately, there exists a range of other tools, such as ProVerif [9] or AVISPA [1], specifically made for verifying security properties of cryptographic protocols. Hence, a separate model could be created to complement our model and to verify properties that are not possible to examine with mCRL2.

However, as described in section 4.1.2, our model does support compromising any of the in-band channels and modifying messages in order to examine how unexpected situations are dealt with. Currently, the only supported modification is exchanging the *NoobId* during the Completion Exchange. In future work, additional modifications could be implemented for examining the behaviour of the protocol in case of non-persistent attacks or unreliable channels. Correspondingly, the number of properties tested with queries could be increased and diversified. Furthermore, as the protocol is still in its development stage, it will likely change in the future, requiring new properties and features to be implemented.

Due to limited time and resources, the state space of tested instances remained fairly small (less than 1 000 000 states) throughout the verification process. In addition to abstracting away most of the irrelevant data, the lists of supported version numbers to be negotiated between, as well as the number of allowed nonces and identifiers to be generated, were strictly limited. For each individual execution of the test cases, the limits were adjusted to be sufficient for successful completion of each exchange, but limited enough to avoid unnecessary resets or packet losses. The number of servers and peers

was kept low due to the same reasons.

One way of possibly improving the efficiency of the verification process would be to use an independent tool, such as *LTSmin* [11, 31], for the the model checking process. It is a high-performance model checker with built-in support for mCRL2, that provides tools for state space generation and minimisation, as well as symbolic, multi-core, sequential and distributed model checking through the *Partitioned Next-State Interface* (PINS).

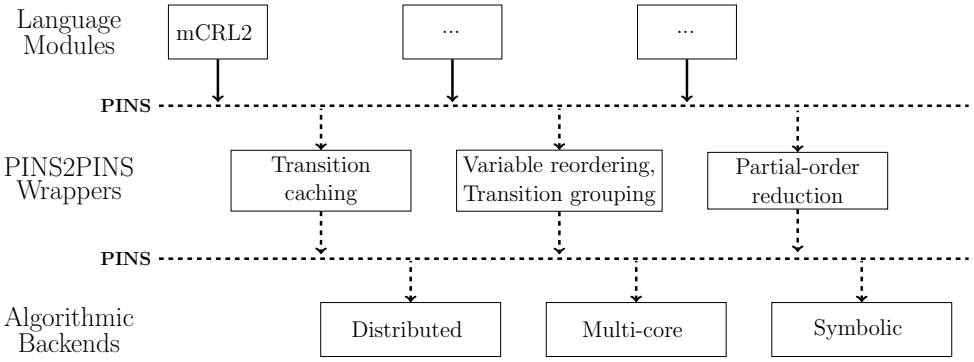

Figure 7.2: LTSmin - Architectural Overview[2]

Comparative experiments have shown that the performance of the LTSmin toolset matches, or often improves, the efficiency of the language-specific tools [11]. As of now, model checking the compiled specification is essentially the only task capable of running in parallel, making the compilation process (mCRL2 specification → LPS → LTS) a bottleneck. Overcoming this issue would improve the speed of simulating large systems with multiple participants.

[2]Adapted and modified from http://ltsmin.utwente.nl/#pins2pins-wrappers

Chapter 8

Conclusions

The expansion of the Internet of Things (IoT) has resulted in an increasing number of new IoT devices to constantly communicate over the network with servers and with other devices. This has created a need for developing new, secure communication protocols to ensure secrecy and to prohibit exploiting the large number of independent devices for e.g. denial-of-service attacks. As a result, formal verification methods are becoming an important part of the system development process. In addition to ensuring the correctness of each individual implementation, the specification itself has to be confirmed to be reliable and secure.

Model checking is a general verification approach that verifies the formal specification of a system by examining a model of it. The process generally consists of creating the model, formalising the required properties and testing whether they hold in the state space of the model. By applying minimisation techniques, such as abstraction and partial order reduction, the state space can often be reduced significantly or at least be delimited to a finite size. Exhaustive exploration of the state space helps detecting obscure mistakes that are easily overlooked by traditional testing methods.

In this thesis, we created a symbolic model of the EAP-NOOB protocol, used for bootstrapping IoT devices, and verified various reachability properties in a multi-server and multi-peer environment. We identified several issues with the previous version of the protocol and proposed some changes made in version 03 of the draft. Major findings relate to the recovery of the protocol after lost or corrupted messages, which could be exploited for denial-of-service attacks. Our model was also used to verify the changes we made and to investigate the behaviour of the protocol in various situations. With the results from both the modelling and the verification process, we improved and clarified the specification and implementation of the protocol.

Throughout the verification process, we compared the results of the model

with the protocol implementation and the test script to minimise the risk of mistakes. However, even though our model implies correct behaviour of the system, we cannot guarantee complete reliability or validity. Instead of seeing the model as proof of correctness, it should be considered as an additional tool for testing and finding flaws. As with any verification method, we should still consider the possibility of unknown issues discovered in the future. Furthermore, the issues we managed to identify were all related to reachability and recovery from errors. Some of these flaws can be exploited by attackers for denial-of-service. We did not verify other security aspects of the cryptographic protocol, leaving room for future work to be done in regards to formal verification of the specification.

Bibliography

[1] Armando, A., Basin, D., Boichut, Y., Chevalier, Y., Compagna, L., Cuellar, J., Hankes, P. D., Heám, P. C., Kouchnarenko, O., Mantovani, J., Mödersheim, S., von Oheimb, D., Rusinowitch, M., Santiago, J., Turuani, M., Viganò, L., and Vigneron, L. The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In *Computer Aided Verification* (Berlin, Heidelberg, 2005), Springer Berlin Heidelberg, pp. 281–285.

[2] Aura, T., and Sethi, M. Nimble out-of-band authentication for EAP (EAP-NOOB). Internet-Draft, draft-aura-eap-noob-02, Internet Engineering Task Force, May 2017. Work in Progress.

[3] Aura, T., and Sethi, M. Nimble out-of-band authentication for EAP (EAP-NOOB). Internet-Draft draft-aura-eap-noob-03, Internet Engineering Task Force, July 2018. Work in Progress.

[4] Baeten, J. C. M. A Brief History of Process Algebra. *Theoretical Computer Science 335*, 2 (May 2005), 131–146.

[5] Baier, C., and Katoen, J.-P. *Principles of Model Checking*. The MIT Press, Cambridge, MA, USA, 2008. Book.

[6] Barker, E., Chen, L., Roginsky, A., and Smid, M. Recommendation for Pair-wise Key Establishment Schemes Using Discrete Logarithm Cryptography. *NIST Special Publication 800-56A Revision 2* (May 2013).

[7] Bergstra, J. A., and Klop, J. W. Process Algebra for Synchronous Communication. *Information and Control 60*, 1 (March 1984), 109–137.

[8] Biere, A., Cimatti, A., Clarke, E., and Zhu, Y. Symbolic Model Checking without BDDs. In *Tools and Algorithms for the Construction*

and Analysis of Systems (Berlin, Heidelberg, 1999), Springer Berlin Heidelberg, pp. 193–207.

[9] Blanchet, B. Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif. *Foundations and Trends in Privacy and Security 1*, 1-2 (October 2016), 1–135.

[10] Blom, S., Fokkink, W., Groote, J. F., van Langevelde, I., Lisser, B., and van de Pol, J. µCRL: A Toolset for Analysing Algebraic Specifications. In *Computer Aided Verification* (Berlin, Heidelberg, 2001), Springer Berlin Heidelberg, pp. 250–254.

[11] Blom, S., van de Pol, J., and Weber, M. LTSmin: Distributed and Symbolic Reachability. In *Computer Aided Verification* (Berlin, Heidelberg, 2010), Springer Berlin Heidelberg, pp. 354–359.

[12] Burch, J., Clarke, E., McMillan, K., Dill, D., and Hwang, L. Symbolic Model Checking: 1020 States and Beyond. *Information and Computation 98* (June 1992), 142–170.

[13] Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P., Papakonstantinou, I., Purbrick, J., and Rodriguez, D. Moving Fast with Software Verification. In *NASA Formal Methods* (Cham, 2015), Springer International Publishing, pp. 3–11.

[14] Clarke, E. M., and Emerson, E. A. Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. In *Logics of Programs* (Berlin, Heidelberg, 1982), Springer Berlin Heidelberg, pp. 52–71.

[15] Clarke, E. M., Henzinger, T. A., and Veith, H. *Introduction to Model Checking*. Springer International Publishing, Cham, 2018, pp. 1–26. In *Handbook of Model Checking*.

[16] Clarke, E. M., Klieber, W., Nováček, M., and Zuliani, P. *Model Checking and the State Explosion Problem*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 1–30. In *Tools for Practical Software Verification*.

[17] Clarke Jr., E. M., Grumberg, O., and Peled, D. A. *Model Checking*. The MIT Press, Cambridge, MA, USA, 1999. Book.

[18] Cranen, S., Groote, J. F., Keiren, J. J. A., Stappers, F. P. M., de Vink, E. P., Wesselink, W., and Willemse, T. A. C. An Overview of the mCRL2 Toolset and Its Recent Advances. In *Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2013), Springer Berlin Heidelberg, pp. 199–213.

[19] Cremers, C., Horvat, M., Hoyland, J., Scott, S., and van der Merwe, T. A Comprehensive Symbolic Analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2017), ACM, pp. 1773–1788.

[20] DeKok, A. The Network Access Identifier. RFC 7542, RFC Editor, May 2015.

[21] Groote, J. F., Mathijssen, A., Reniers, M., Usenko, Y., and van Weerdenburg, M. The Formal Specification Language mCRL2. In *Methods for Modelling Software Systems (MMOSS)* (Dagstuhl, Germany, 2007), Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI).

[22] Groote, J. F., Mathijssen, A., van Weerdenburg, M., and Usenko, Y. From µCRL to mCRL2: Motivation and Outline. *Electronic Notes in Theoretical Computer Science 162* (September 2006), 191–196.

[23] Groote, J. F., and Mousavi, M. R. *Modelling and Analysis of Communicating Systems*. The MIT Press, Cambridge, MA, USA, 2014. Book.

[24] Groote, J. F., and Ponse, A. The Syntax and Semantics of μCRL. In *Algebra of Communicating Processes* (London, 1995), Springer London, pp. 26–62.

[25] Halpern, J. Y., and Moses, Y. Knowledge and Common Knowledge in a Distributed Environment. *Journal of the ACM 37*, 3 (July 1990), 549–587.

[26] Hennessy, M., and Milner, R. On Observing Nondeterminism and Concurrency. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming* (Berlin, Heidelberg, 1980), Springer-Verlag, pp. 299–309.

[27] Hennessy, M., and Milner, R. Algebraic Laws for Nondeterminism and Concurrency. *Journal of the ACM 32*, 1 (January 1985), 137–161.

[28] Information Sciences Institute, U. o. S. C. Transmission Control Protocol. RFC 793, RFC Editor, September 1981.

[29] Jones, M. JSON Web Key (JWK). RFC 7517, RFC Editor, May 2015.

[30] Josefsson, S. The Base16, Base32, and Base64 Data Encodings. RFC 4648, RFC Editor, October 2006.

[31] Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., and van Tom Dijk. LTSmin: High-Performance Language-Independent Model Checking. In *Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2015), Springer Berlin Heidelberg, pp. 692–707.

[32] Knuth, D. E., and Bendix, P. B. Simple Word Problems in Universal Algebras. In *Computational Problems in Abstract Algebra*. Pergamon, 1970, pp. 263–297.

[33] Kozen, D. Results on the Propositional µ-calculus. *Theoretical Computer Science 27*, 3 (1983), 333–354.

[34] Krawczyk, H., Bellare, M., and Canetti, R. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, RFC Editor, February 1997.

[35] Kuo-Chung, T. Definitions and Detection of Deadlock, Livelock, and Starvation in Concurrent Programs. In *1994 Internatonal Conference on Parallel Processing Vol. 2* (1994), IEEE, pp. 69–72.

[36] Langley, A., Hamburg, M., and Turner, S. Elliptic Curves for Security. RFC 7748, RFC Editor, January 2016.

[37] Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., and Deardeuff, M. How Amazon Web Services Uses Formal Methods. *Communications of the ACM 58*, 4 (March 2015), 66–73.

[38] Prakash, S. T. Enhancements to Secure Bootstrapping of Smart Appliances. Master's thesis, Aalto University, August 2017. `http://urn.fi/URN:NBN:fi:aalto-201709046881`.

[39] Queille, J. P., and Sifakis, J. Specification and Verification of Concurrent Systems in CESAR. In *International Symposium on Programming* (Berlin, Heidelberg, 1982), Springer Berlin Heidelberg, pp. 337–351.

[40] Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.3. Internet-Draft draft-ietf-tls-tls13-28, RFC Editor, March 2018.

[41] Seetarama, R. M. Secure Device Bootstrapping with the Nimble Out of Band Authentication Protocol. Master's thesis, Aalto University, May 2017. `http://urn.fi/URN:NBN:fi:aalto-201706135412`.

[42] Sethi, M. *Security for Ubiquitous Internet-Connected Smart Objects*. Doctoral dissertation, Aalto University, December 2016. `http://urn.fi/URN:ISBN:978-952-60-7224-1`.

[43] Sethi, M., Oat, E., Di Francesco, M., and Aura, T. Secure Bootstrapping of Cloud-managed Ubiquitous Displays. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing* (New York, NY, USA, 2014), ACM, pp. 739–750.

[44] Technische Universiteit Eindhoven. Introduction to mCRL2. Webpage. `https://www.mcrl2.org/web/user_manual/introduction.html`. Accessed 18.5.2018.

[45] Vollbrecht, J., Carlson, J. D., Blunk, L., Aboba, B. D., and Levkowetz, H. Extensible Authentication Protocol (EAP). RFC 3748, RFC Editor, June 2004.

Appendix A

Changes in mCRL2

During the modelling process, we discovered various minor defects and lacks of features in the modelling language, requiring simple workarounds to implement specific parts of the protocol. However, we also detected a bug in the type checking algorithm, causing exponential compilation times when using user-declared type aliases. The bug was reported and later fixed by the development team. In this section, we describe the causes and consequences of said bug.

A.1 Slow Type Checking

To make the model comparable to the protocol specification, we used the same variable names as defined in the draft [3]. However, as described in chapter 4, abstraction is one of the main methods we used for reducing the state space of the model. As a result, many of the data types, such as the identifiers and the nonces, were replaced by natural numbers.

Fortunately, mCRL2 allows declaring sort aliases, creating pairwise equal types to be used in the specification. For example, declaring a custom data type $PeerId$ as a natural number (Nat)

```
% Define an alias mapping PeerId to Nat
sort PeerId = Nat;
```

Listing A.1: Type alias for *PeerId*

replaces all consequent occurrences of $PeerId$ with Nat during the compilation process. Our intention was to use this feature to even further improve the readability of the model by introducing an alias for each individual type. However, in the current stable release version (201409.0) of the mCRL2

toolset, using multiple custom types slows down the compilation process exponentially.

Consider the following example:

```
1  % Map several custom data types to natural numbers
2  sort
3     A_t = Nat; B_t = Nat; C_t = Nat; D_t = Nat;
4     E_t = Nat; F_t = Nat; G_t = Nat;
5
6  % Create a new structure that consists of custom data
7  % types
8  S_t = struct s(
9     A:A_t, B:B_t, C:C_t, D:D_t, E:E_t, F:F_t, G:G_t
10  % A:Nat, B:Nat, C:Nat, D:Nat, E:Nat, F:Nat, G:Nat
11 );
12
13 act
14    a;
15
16 proc
17    P = a;
18
19 init
20    P;
```

Listing A.2: Using type aliases in structures

Intuitively, each occurrence of a custom data type on row 8 should be replaced by *Nat*, making rows 8 and 9 equivalent. However, compiling the specification using version 201409.0 on an Intel(R) Core(TM) i5-7300U (2.60GHz) CPU takes more than 240 seconds, whereas switching lines 8 and 9 reduces the compilation time to a fraction of a second.

A bug report[1] was submitted to the mCRL2 mailing list[2] and later verified a by the developers, who identified it as a mistake in creating unique normal forms of structures using the Knuth-Bendix completion [32]. In the above example, the compiler would create exponentially many variants of the structure, replacing each custom data type with a rewrite rule (as shown in listing A.3).

[1] https://github.com/mCRL2org/mCRL2/issues/1456
[2] https://listserver.tue.nl/pipermail/mcrl2-users/

```
struct s(A:Nat, B:B_t, C:C_t, D:D_t, E:E_t, F:F_t, G:G_t) = S_t
struct s(A:A_t, B:Nat, C:C_t, D:D_t, E:E_t, F:F_t, G:G_t) = S_t
struct s(A:A_t, B:B_t, C:Nat, D:D_t, E:E_t, F:F_t, G:G_t) = S_t
  ⋮
struct s(A:Nat, B:Nat, C:Nat, D:Nat, E:Nat, F:Nat, G:Nat) = S_t
```

Listing A.3: Exponential growth of structures

Internally, even more duplicates were created, making compiling larger structures infeasible. The issue was submitted to the bug tracker of the project and fixed shortly after. Compiling the specification after applying change set 15361 (version 201707.1.15361 or later) removes the issue and makes creating custom data types equal to predefined types.

Appendix B

Model Code

```
sort
    % === Data types of communication and cryptographic functions === %

    % States 0 to 4, no state, error state (user action required)
    State_t = struct s0 | s1 | s2 | s3 | s4 | no_state | error ? is_error;

    % Types 0 to 8, no type, identity,  failure, success
    Type_t = struct t0 | t1 | t2 | t3 | t4 | t5 | t6 | t7 | t8 | no_type | id | failure | success;

    % Data types
    Ver_t          = Nat;                 % Version
    Ver_l          = List(Ver_t);         % List of versions
    PeerId_t       = Nat;                 % Peer ID
  % Realm_t        = Nat;                 % Realm
    PK_t           = Nat;                 % Public key
    Cryptosuite_t  = Nat;                 % Cryptosuite
    Cryptosuite_l  = List(Cryptosuite_t); % List of cryptosuites
    Dir_t          = Nat;                 % Direction
    N_t            = Nat;                 % Nonce
  % SleepTime_t    = Nat;                 % Sleep time
    Noob_t         = Nat;                 % Noob
    NoobId_t       = Nat;                 % Noob ID

    % Server or Peer information
    Info_t = struct no_info ? is_noinfo
        | info        ? is_validinfo    % Valid server information
        | invalidinfo ? is_invalidinfo  % Invalid server information
    ;

    % Error codes
    Error_t = struct
      % | E1001  % Invalid NAI
      % | E1002  % Invalid message structure
        | E1003  % Invalid data
        | E1004  % Unexpected message type
        | E1005  % Unexpected peer identifier
        | E1006  % Unrecognised OOB message identifier
      % | E1007  % Invalid ECDH key
      % | E2001  % Unwanted peer
        | E2002  % State mismatch, user action required
        | E3001  % No mutually supported protocol version
        | E3002  % No mutually supported cryptosuite
        | E3003  % No mutually supported OOB direction
        | E4001  % MAC verification failure
      % | E5001  % Application-specific error
        | E5002  % Invalid server info
      % | E5003  % Invalid server URL
        | E5004  % Invalid peer info
    ;

    % Cryptographic hash Hoob
    Hoob_t = struct H(
        Dir:Nat, Vers:Ver_l, Verp:Nat, PeerId:Nat, Cryptosuites:Cryptosuite_l, Dirs:Nat, ServerInfo:Info_t,
        Cryptosuitep:Nat, Dirp:Nat, PeerInfo:Info_t, PKs:Nat, Ns:Nat, PKp:Nat, Np:Nat, Noob:Nat)
    ;
```

```
56
57      % Cryptographic MAC
58      MAC_t = struct HMAC(K:K_t,
59          Dir:Nat, Vers:Ver_l, Verp:Nat, PeerId:Nat, Cryptosuites:Cryptosuite_l, Dirs:Nat, ServerInfo:Info_t,
60          Cryptosuitep:Nat, Dirp:Nat, PeerInfo:Info_t, PKs:Nat, Ns:Nat, PKp:Nat, Np:Nat, Noob:Nat)
61      ;
62
63      % ECDH key derivation (commutative)
64      ECDH_t = struct ECDH(PK1:Nat, PK2:Nat);
65
66      % Derived keys (see Table 3)
67      K_t = struct no_key
68          | Completion(slice:Nat, Z:ECDH_t, Np:Nat, Ns:Nat, Noob:Nat)
69          | RekeyingECDH(slice:Nat, Z:ECDH_t, Np2:Nat, Ns2:Nat, Kz:K_t)
70          | Rekeying(slice:Nat, Kz:K_t, Np2:Nat, Ns2:Nat)
71      ;
72
73      % Data sent between peer and server
74      Data_t = struct empty_d ? is_empty
75          % Type 1-8 requests and responses
76          | req1(Vers:Ver_l, Cryptosuites:Cryptosuite_l, Dirs:Nat, ServerInfo:Info_t) ? is_req1
77          | res1(Verp:Nat, Cryptosuitep:Nat, Dirp:Nat, PeerInfo:Info_t) ? is_res1
78          | req2(PKs:Nat, Ns:Nat) ? is_req2
79          | res2(PKp:Nat, Np:Nat) ? is_res2
80          | req3 ? is_req3
81          | res3 ? is_res3
82          | req4(NoobId:Nat, MACs:MAC_t) ? is_req4
83          | res4(MACp: MAC_t) ? is_res4
84          | req5(Vers:Ver_l, Cryptosuites:Cryptosuite_l) ? is_req5
85          | res5(Verp:Nat, Cryptosuitep:Nat) ? is_res5
86          | req6(PKs2:Nat, Ns2:Nat) ? is_req6
87          | res6(PKp2:Nat, Np2:Nat) ? is_res6
88          | req7(MACs2:MAC_t) ? is_req7
89          | res7(MACp2:MAC_t) ? is_res7
90          | req8 ? is_req8
91          | res8(NoobId:Nat) ? is_res8
92          % Error message
93          | err(ErrorCode:Error_t) ? is_error_msg
94          % Invalid data
95          | invalid ? is_invalid
96      ;
97
98      % === Internal data types for peer and server implementation === %
99
100     % Initialization data for peer
101     StaticPeerData_t = struct empty_sp
102         | static_peer_data(Vers:Ver_l, Cryptosuites:Cryptosuite_l, Dirs:Nat, PeerInfo:Info_t, PK:Nat, PK2:Nat)
103     ;
104
105     % Initialization data for server
106     StaticServerData_t = struct empty_s
107         | static_server_data(Vers:Ver_l, Cryptosuites:Cryptosuite_l, Dirs:Nat, ServerInfo:Info_t, PK:Nat, PK2:Nat)
108     ;
109
110     % Association database interface at server, updating fields and querying peer entries
111     DBInterface_t = struct empty_dbi
112         | Noob(Noob:Nat) ? is_noob
113         | NoobId(NoobId:Nat) ? is_noobid
114         | Nonce(N:Nat) ? is_nonce
115         | Type(Type:Type_t) ? is_type
116         | Keys(Kms:K_t, Kmp:K_t) ? is_key
117         | DBEntry(Verp:Nat, Cryptosuitep:Nat, Dirp:Nat, PeerInfo:Info_t, PKp:Nat, Np:Nat, Ns:Nat, Noob:Nat,
118             Type:Type_t, Kms:K_t, Kmp:K_t) ? is_dbentry
119     ;
120
121     % Association database entries at server (keyed by PeerId:Nat)
122     State_e        = Nat -> State_t;
123     Verp_e         = Nat -> Nat;
124     Cryptosuitep_e = Nat -> Nat;
125     Dirp_e         = Nat -> Nat;
126     PeerInfo_e     = Nat -> Info_t;
127     PKp_e          = Nat -> Nat;
128     Np_e           = Nat -> Nat;
129     Ns_e           = Nat -> Nat;
130     Noob_e         = Nat -> Nat;
131     OOB_e          = Nat -> Nat;
132     Type_e         = Nat -> Type_t;
133     Kms_e          = Nat -> K_t;
134     Kmp_e          = Nat -> K_t;
135
136 act
137     % SEND_xxx    = send to channel multiaction
138     % SEND_xxx_0  = send to channel, action on sender side
```

```
139     % SEND_xxx_I  = send to channel, action on channel side
140     % RECV_xxx    = receive from channel multiaction
141     % RECV_xxx_O  = receive from channel, action on sender side
142     % RECV_xxx_I  = receive from channel, action on channel side
143
144     % == EAP-Response/Identity == %
145     SEND_EAP_RES_ID,
146     RECV_EAP_RES_ID,
147     SEND_EAP_RES_ID_O,
148     SEND_EAP_RES_ID_I,
149     RECV_EAP_RES_ID_O,
150     RECV_EAP_RES_ID_I: PeerId_t # State_t;
151
152     % == EAP-Request/EAP-NOOB == %
153     SEND_EAP_REQ,
154     RECV_EAP_REQ,
155     SEND_EAP_REQ_O,
156     SEND_EAP_REQ_I,
157     RECV_EAP_REQ_O,
158     RECV_EAP_REQ_I: Type_t # PeerId_t # Data_t;
159
160     % == EAP-Response/EAP-NOOB == %
161     SEND_EAP_RES,
162     RECV_EAP_RES,
163     SEND_EAP_RES_O,
164     SEND_EAP_RES_I,
165     RECV_EAP_RES_O,
166     RECV_EAP_RES_I: Type_t # PeerId_t # Data_t;
167
168     % == EAP-Failure == %
169     SEND_EAP_FAIL;
170     RECV_EAP_FAIL;
171     SEND_EAP_FAIL_O;
172     SEND_EAP_FAIL_I;
173     RECV_EAP_FAIL_O;
174     RECV_EAP_FAIL_I;
175
176     % == EAP-Success == %
177     SEND_EAP_SUCC;
178     RECV_EAP_SUCC;
179     SEND_EAP_SUCC_O;
180     SEND_EAP_SUCC_I;
181     RECV_EAP_SUCC_O;
182     RECV_EAP_SUCC_I;
183
184     % == OOB message in peer-to-server (P2S) and server-to-peer (S2P) direction == %
185     SEND_OOB_P2S,
186     RECV_OOB_P2S,
187     SEND_OOB_S2P,
188     RECV_OOB_S2P,
189     SEND_OOB_P2S_O,
190     SEND_OOB_P2S_I,
191     SEND_OOB_S2P_O,
192     SEND_OOB_S2P_I,
193     RECV_OOB_P2S_O,
194     RECV_OOB_P2S_I,
195     RECV_OOB_S2P_O,
196     RECV_OOB_S2P_I: PeerId_t # Noob_t # Hoob_t;
197
198     % == Mobility/Timeout/Failure == %
199     MOBILITY_TIMEOUT_FAILURE;
200
201     % == User Reset == %
202     USER_RESET;
203
204     % == Query server and peer state, error status during testing and verification == %
205     SERV_STATE,
206     PEER_STATE: PeerId_t # State_t;
207     LOG_ERROR:  Error_t;
208
209     % == Notify that a message was lost. For testing and verification == %
210     MESSAGE_LOST: Type_t;
211
212     % == Generate random PeerId, Noob, nonces == %
213     % Multiaction for any random number generation
214     RNG_MA: Nat;
215     % Generator and server/peer actions
216     NEW_PEERID_RNG,
217     NEW_PEERID:    PeerId_t;
218     NEW_NOOB_RNG,
219     NEW_NOOB:      Noob_t;
220     NEW_NONCE_RNG,
221     NEW_NONCE:     N_t;
```

```
222     % Can't generate more values
223     MAX_PEERIDS_REACHED;
224     MAX_NOOBS_REACHED;
225     MAX_NONCES_REACHED;
226
227     % == Query and update the association database at server == %
228     % _MA = multiaction, _DB = action at database, no _x = action at server
229     QUERY_STATE_MA,
230     QUERY_STATE_DB,
231     QUERY_STATE:           PeerId_t # State_t # Type_t;
232     QUERY_DATA_MA,
233     QUERY_DATA_DB,
234     QUERY_DATA:            PeerId_t # DBInterface_t;
235     QUERY_FAILED_OOBS_MA,
236     QUERY_FAILED_OOBS_DB,
237     QUERY_FAILED_OOBS:     PeerId_t # Nat;
238     UPDATE_FAILED_OOBS_MA,
239     UPDATE_FAILED_OOBS_DB,
240     UPDATE_FAILED_OOBS:    PeerId_t # Nat;
241     UPDATE_STATE_MA,
242     UPDATE_STATE_DB,
243     UPDATE_STATE:          PeerId_t # State_t;
244     UPDATE_DATA_MA,
245     UPDATE_DATA_DB,
246     UPDATE_DATA:           PeerId_t # Data_t;
247     UPDATE_NONCE_MA,
248     UPDATE_NONCE_DB,
249     UPDATE_NONCE:          PeerId_t # DBInterface_t;
250     UPDATE_TYPE_MA,
251     UPDATE_TYPE_DB,
252     UPDATE_TYPE:           PeerId_t # DBInterface_t;
253     UPDATE_KEY_MA,
254     UPDATE_KEY_DB,
255     UPDATE_KEY:            PeerId_t # DBInterface_t;
256
257     % == Reset database for a PeerId == %
258     RESET_DATABASE_MA,
259     RESET_DATABASE_DB,
260     RESET_DATABASE:    PeerId_t;
261
262 map
263     % Static data structures
264     static_peer_data: StaticPeerData_t;   % Peer data
265     static_serv_data: StaticServerData_t; % Server data
266
267     % Constants
268     max_peers:       Nat; % Max PeerIds generated
269     max_noobs:       Nat; % Max Noob values generated
270     max_nonces:      Nat; % Max nonces generated
271     max_oob_retries: Nat; % Application-specific number of invalid oob messages
272
273     % Database entry at the server: db contains one of each field per peer
274     state:        State_e;
275     verp:         Verp_e;
276     cryptosuitep: Cryptosuitep_e;
277     dirp:         Dirp_e;
278     peerinfo:     PeerInfo_e;
279     pkp:          PKp_e;
280     np:           Np_e;
281     ns:           Ns_e;
282     noob:         Noob_e;
283     oobretries:   OOB_e;
284     type:         Type_e;
285     kms:          Kms_e;
286     kmp:          Kmp_e;
287
288 var
289     a, a',
290     b, b' ,
291     n, n': Nat;
292     l, l': List(Nat);
293
294 eqn
295     % Initialize static data structures at peer and server
296     %   Version(s), Cryptosuite(s), Dir, PeerInfo, PK, PK2
297     static_peer_data = static_peer_data([1], [1], 2, info, 1, 2);
298     %   Version(s), Cryptosuite(s), Dir, ServInfo, PK, PK2
299     static_serv_data = static_server_data([1], [1], 2, info, 3, 4);
300
301     % Maximum number of PeerIds, Noob values, nonce values in the model
302     max_peers  = 1; % >=1
303     max_noobs  = 2; % >=2 if Dirp=1 or Dirp=2, >=3 if Dirp=3
304     max_nonces = 4; % >=4
```

```
305
306     % Maximum number of failed OOB messages before going back to state 0
307     max_oob_retries = 2;
308
309     % Default values for database at the server
310     state(n)        = no_state;
311     verp(n)         = 0;
312     cryptosuitep(n) = 0;
313     dirp(n)         = 0;
314     peerinfo(n)     = no_info;
315     pkp(n)          = 0;
316     np(n)           = 0;
317     ns(n)           = 0;
318     noob(n)         = 0;
319     oobretries(n)   = 0;
320     type(n)         = no_type;
321     kms(n)          = no_key;
322     kmp(n)          = no_key;
323
324     % ECDH is commutative
325     ECDH(a, b) == ECDH(a', b') = (a == a' && b == b') || (a == b' && b == a');
326
327 proc
328     % In-band channel: Peer to Server
329     %   compromised : true, if the channel is controlled by an attacker, false otherwise
330     PeerToServerChannel(compromised:Bool) =
331         sum PeerId:PeerId_t, State:State_t . (
332             SEND_EAP_RES_ID_I(PeerId, State)
333             . (RECV_EAP_RES_ID_O(PeerId, State) + MESSAGE_LOST(id))
334             . PeerToServerChannel(compromised)
335         )
336         + sum PeerId:PeerId_t, data:Data_t, type:Type_t . (
337             SEND_EAP_RES_I(type, PeerId, data)
338             % Modify values if the channel is compromised
339             . (compromised) -> (
340                 (type == t8) -> (
341                     % Spoof NoobId
342                     RECV_EAP_RES_O(type, PeerId, res8(NoobId(data) + 20))
343                     . PeerToServerChannel(false)
344                 )
345                 <> (RECV_EAP_RES_O(type, PeerId, data) + MESSAGE_LOST(type))
346                 . PeerToServerChannel(compromised)
347             )
348             <> (RECV_EAP_RES_O(type, PeerId, data) + MESSAGE_LOST(type))
349             . PeerToServerChannel(compromised)
350         )
351     ;
352
353     % In-band channel: Server to Peer
354     %   compromised : true, if the channel is controlled by an attacker, false otherwise
355     ServerToPeerChannel(compromised:Bool) =
356         sum PeerId:PeerId_t, data:Data_t, type:Type_t . (
357             SEND_EAP_REQ_I(type, PeerId, data)
358             % Modify values if the channel is compromised
359             . (compromised) -> (
360                 (type == t4) -> (
361                     % Spoof NoobId
362                     RECV_EAP_REQ_O(type, PeerId, req4(NoobId(data) + 10, MACs(data)))
363                     . ServerToPeerChannel(false)
364                 )
365                 <> (RECV_EAP_REQ_O(type, PeerId, data) + MESSAGE_LOST(type))
366                 . ServerToPeerChannel(compromised)
367             )
368             <> (RECV_EAP_REQ_O(type, PeerId, data) + MESSAGE_LOST(type))
369             . ServerToPeerChannel(compromised)
370         )
371         + SEND_EAP_SUCC_I
372         . (RECV_EAP_SUCC_O + MESSAGE_LOST(success))
373         . ServerToPeerChannel(compromised)
374         + SEND_EAP_FAIL_I
375         . (RECV_EAP_FAIL_O + MESSAGE_LOST(failure))
376         . ServerToPeerChannel(compromised)
377     ;
378
379     % OOB Channel: Peer to Server
380     PeerToServerOOBChannel =
381         sum PeerId:PeerId_t, Noob:Noob_t, Hoob:Hoob_t . (
382             SEND_OOB_P2S_I(PeerId, Noob, Hoob)
383             . RECV_OOB_P2S_O(PeerId, Noob, Hoob)
384             . PeerToServerOOBChannel
385         )
386     ;
387
```

```
388     % OOB Channel: Server to Peer
389     ServerToPeerOOBChannel =
390         sum PeerId:PeerId_t, Noob:Noob_t, Hoob:Hoob_t . (
391             SEND_OOB_S2P_I(PeerId, Noob, Hoob)
392             . RECV_OOB_S2P_O(PeerId, Noob, Hoob)
393             . ServerToPeerOOBChannel
394         )
395     ;
396
397     % Random number generator
398     %   PeerId : Next PeerId to generate
399     %   Noob   : Next Noob to generate
400     %   N      : Next nonce to generate
401     Rng(PeerId:PeerId_t, Noob:Noob_t, N:N_t) =
402         (PeerId <= max_peers) -> (
403             NEW_PEERID_RNG(PeerId)
404             . (PeerId == max_peers) -> (
405                 MAX_PEERIDS_REACHED
406                 . Rng(PeerId+1, Noob, N)
407             ) <> Rng(PeerId+1, Noob, N)
408         ) <> MAX_PEERIDS_REACHED . Rng(PeerId, Noob, N)
409         + (Noob <= max_noobs) -> (
410             NEW_NOOB_RNG(Noob)
411             . (Noob == max_noobs) -> (
412                 MAX_NOOBS_REACHED
413                 . Rng(PeerId, Noob+1, N)
414             ) <> Rng(PeerId, Noob+1, N)
415         ) <> MAX_NOOBS_REACHED . Rng(PeerId, Noob, N)
416         + (N <= max_nonces) -> (
417             NEW_NONCE_RNG(N)
418             . (N == max_nonces) -> (
419                 MAX_NONCES_REACHED
420                 . Rng(PeerId, Noob, N+1)
421             ) <> Rng(PeerId, Noob, N+1)
422         ) <> MAX_NONCES_REACHED . Rng(PeerId, Noob, N)
423     ;
424
425     % Server database
426     Database(state:State_e, verp:Verp_e, cryptosuitep:Cryptosuitep_e, dirp:Dirp_e, peerinfo:PeerInfo_e, pkp:PKp_e,
427         np:Np_e, ns:Ns_e, noob:Noob_e, oobretries:OOB_e, type:Type_e, kms:Kms_e, kmp:Kmp_e) =
428         sum PeerId:PeerId_t . (
429             sum State:State_t . (
430                 QUERY_STATE_DB(PeerId, state(PeerId), type(PeerId))
431                 . Database()
432                 + UPDATE_STATE_DB(PeerId, State)
433                 . SERV_STATE(PeerId, State)
434                 . Database(
435                     state = state[PeerId->State]
436                 )
437             )
438             + QUERY_FAILED_OOBS_DB(PeerId, oobretries(PeerId))
439             + QUERY_DATA_DB(PeerId, DBEntry(
440                 verp(PeerId), cryptosuitep(PeerId), dirp(PeerId), peerinfo(PeerId), pkp(PeerId), np(PeerId),
441                 ns(PeerId), noob(PeerId), type(PeerId), kms(PeerId), kmp(PeerId)
442             ))
443             . Database()
444             + sum OobRetries:Nat . (
445                 UPDATE_FAILED_OOBS_DB(PeerId, OobRetries)
446                 . Database(
447                     oobretries = oobretries[PeerId->OobRetries]
448                 )
449             )
450             + sum data:Data_t . (
451                 UPDATE_DATA_DB(PeerId, data) . (
452                 % Type 1 response
453                 (is_res1(data)) -> (
454                     Database(
455                         verp         = verp[PeerId->Verp(data)],
456                         cryptosuitep = cryptosuitep[ PeerId->Cryptosuitep(data)],
457                         dirp         = dirp[PeerId->Dirp(data)],
458                         peerinfo     = peerinfo[PeerId->PeerInfo(data)]
459                     )
460                 )
461                 % Type 2 response
462                 <> (is_res2(data)) -> (
463                     Database(
464                         pkp = pkp[PeerId->PKp(data)],
465                         np  = np[PeerId->Np(data)]
466                     )
467                 )
468                 % Type 5 response
469                 <> (is_res5(data)) -> (
470                     Database(
```

```
471                         verp         = verp[PeerId->Verp(data)],
472                         cryptosuitep = cryptosuitep[PeerId->Cryptosuitep(data)]
473                     )
474                 )
475                 <> (is_res6(data)) -> (
476                     % Type 6 response (w/ PKp2)
477                     (PKp2(data) != 0) -> (
478                         Database(
479                             pkp = pkp[PeerId->PKp2(data)],
480                             np  = np[PeerId->Np2(data)]
481                         )
482                     )
483                     % Type 6 response (w/o PKp2)
484                     <> (PKp2(data) == 0) -> (
485                         Database(
486                             np = np[PeerId->Np2(data)]
487                         )
488                     )
489                 )
490             ))
491             + sum data:DBInterface_t . (
492                 UPDATE_NONCE_DB(PeerId, data) . (
493                     % Server nonce update
494                     (is_nonce(data)) -> (
495                         Database(
496                             ns = ns[PeerId->N(data)]
497                         )
498                     )
499                     % Server noob update
500                     <> (is_noob(data)) -> (
501                         Database(
502                             noob = noob[PeerId->Noob(data)]
503                         )
504                     )
505                 )
506                 % Ongoing exchange update
507                 + UPDATE_TYPE_DB(PeerId, data)
508                 . (is_type(data)) -> (
509                     Database(
510                         type = type[PeerId->Type(data)]
511                     )
512                 )
513                 % Key update
514                 + UPDATE_KEY_DB(PeerId, data)
515                 . (is_key(data)) -> (
516                     Database(
517                         kms = kms[PeerId->Kms(data)],
518                         kmp = kmp[PeerId->Kmp(data)]
519                     )
520                 )
521             )
522             % Allow Mobility/Timeout/Failure to happen for the server
523             + (state(PeerId) == s4 && type(PeerId) == no_type) -> (
524                 SERV_STATE(PeerId, s4)
525                 . Database()
526                 + MOBILITY_TIMEOUT_FAILURE
527                 . SERV_STATE(PeerId, s3)
528                 . Database(
529                     state = state[PeerId->s3]
530                 )
531             )
532             % Reset database values for a given PeerId
533             + RESET_DATABASE_DB(PeerId)
534             . SERV_STATE(PeerId, s0)
535             . Database(
536                 state        = state[PeerId->no_state],
537                 verp         = verp[PeerId->0],
538                 cryptosuitep = cryptosuitep[PeerId->0],
539                 dirp         = dirp[PeerId->0],
540                 peerinfo     = peerinfo[PeerId->no_info],
541                 pkp          = pkp[PeerId->0],
542                 np           = np[PeerId->0],
543                 ns           = ns[PeerId->0],
544                 noob         = noob[PeerId->0],
545                 oobretries   = oobretries[PeerId->0],
546                 type         = type[PeerId->no_type],
547                 kms          = kms[PeerId->no_key],
548                 kmp          = kmp[PeerId->no_key]
549             )
550         )
551     ;
552
553     % EAP Server
```

```
554     %   sd    : Server data
555     %   noobs : Generated Noob values
556     Server(sd:StaticServerData_t, noobs:List(Nat)) =
557         sum PeerId:PeerId_t, Peer_State:State_t . (
558             RECV_EAP_RES_ID_I(PeerId, Peer_State) . (
559                 % Initial Exchange
560                 (PeerId == 0) -> (
561                     sum PeerId:PeerId_t . (
562                         NEW_PEERID(PeerId)
563                         . SEND_EAP_REQ_0(t1, PeerId, req1(Vers(sd), Cryptosuites(sd), Dirs(sd), ServerInfo(sd)))
564                         . UPDATE_TYPE(PeerId, Type(t1))
565                     )
566                 )
567                 % Other exchanges
568                 + (PeerId != 0) -> (
569                     sum Server_State:State_t, Type:Type_t . (
570                         QUERY_STATE(PeerId, Server_State, Type) . (
571                             (Server_State != no_state) -> (
572                                 % Waiting Exchange
573                                 (Peer_State == s1 && Server_State == s1) -> (
574                                     SEND_EAP_REQ_0(t3, PeerId, req3)
575                                     . UPDATE_TYPE(PeerId, Type(t3))
576                                 )
577                                 % Completion Exchange (peer-to-server)
578                                 <> (Peer_State == s1 && Server_State == s2) -> (
579                                     sum pd:DBInterface_t . (
580                                         QUERY_DATA(PeerId, pd)
581                                         . UPDATE_KEY(PeerId, Keys(
582                                             Completion(192, ECDH(PK(sd), PKp(pd)), Np(pd), Ns(pd), Noob(pd)),
583                                             Completion(224, ECDH(PK(sd), PKp(pd)), Np(pd), Ns(pd), Noob(pd))
584                                         ))
585                                     )
586                                     . sum pd:DBInterface_t . (
587                                         QUERY_DATA(PeerId, pd)
588                                         . SEND_EAP_REQ_0(t4, PeerId,  req4(
589                                             Noob(pd), HMAC(Kms(pd), 2, Vers(sd), Verp(pd), PeerId, Cryptosuites(sd),
590                                             Dirs(sd), ServerInfo(sd), Cryptosuitep(pd), Dirp(pd), PeerInfo(pd), PK(sd),
591                                             Ns(pd), PKp(pd), Np(pd), Noob(pd))
592                                         ))
593                                     )
594                                     . UPDATE_TYPE(PeerId, Type(t4))
595                                 )
596                                 % Completion Exchange (server-to-peer)
597                                 <> (Peer_State == s2 && (Server_State == s1 || Server_State == s2)) -> (
598                                     SEND_EAP_REQ_0(t8, PeerId, req8)
599                                     . UPDATE_TYPE(PeerId, Type(t8))
600                                 )
601                                 % Reconnect Exchange
602                                 <> (Peer_State == s3 && (Server_State == s3 || Server_State == s4)) -> (
603                                     SEND_EAP_REQ_0(t5, PeerId, req5(Vers(sd), Cryptosuites(sd)))
604                                     . UPDATE_TYPE(PeerId, Type(t5))
605                                 )
606                                 % Drop old messages
607                                 <> (Server_State == s4) -> (
608                                     MESSAGE_LOST(Type)
609                                 )
610                                 % Check for state mismatch
611                                 <> ((Server_State == s1 && (Peer_State == s3 || Peer_State == s4)) ||
612                                     (Server_State == s2 && (Peer_State == s3 || Peer_State == s4)) ||
613                                     (Server_State == s3 && (Peer_State == s1 || Peer_State == s2)) ) -> (
614                                        % Send error E2002
615                                        SEND_EAP_REQ_0(t0, PeerId, err(E2002))
616                                        . Server_Error(no_type, PeerId, E2002)
617                                    )
618                             )
619                             % Received unknown PeerId
620                             <> (Server_State == no_state) -> (
621                                 % Send error E1005
622                                 SEND_EAP_REQ_0(t0, PeerId, err(E1005))
623                                 . LOG_ERROR(E1005)
624                                 . Server(sd, noobs)
625                             )
626                         )
627                     )
628                 )
629             ) . Server(sd, noobs)
630             + sum PeerId:PeerId_t, type:Type_t, data:Data_t . (
631                 RECV_EAP_RES_I(type, PeerId, data) . (
632                     % Receive error message
633                     (type == t0) -> (
634                         sum pd:DBInterface_t . (
635                             QUERY_DATA(PeerId, pd)
636                             . Server_Error(Type(pd), PeerId, ErrorCode(data))
```

```
637                         . Server(sd, noobs)
638                     )
639                 )
640                 % Invalid data
641                 <> (is_invalid(data)) -> (
642                     sum pd:DBInterface_t . (
643                         QUERY_DATA(PeerId, pd)
644                         . SEND_EAP_REQ_0(t0, PeerId, err(E1003))
645                         . Server_Error(Type(pd), PeerId, E1003)
646                         . Server(sd, noobs)
647                     )
648                 )
649                 <> sum Type:Type_t, State:State_t . (
650                     QUERY_STATE(PeerId, State, Type)
651                     % Unexpected message type
652                     . (Type != type) -> (
653                         SEND_EAP_REQ_0(t0, PeerId, err(E1004))
654                         . Server_Error(Type, PeerId, E1004)
655                         . Server(sd, noobs)
656                     )
657                     % Initial Exchange
658                     <> (type == t1) -> (
659                         % Invalid peer info
660                         (is_invalidinfo(PeerInfo(data))) -> (
661                             sum pd:DBInterface_t . (
662                                 QUERY_DATA(PeerId, pd)
663                                 . SEND_EAP_REQ_0(t0, PeerId, err(E5004))
664                                 . Server_Error(Type(pd), PeerId, E5004)
665                             )
666                         )
667                         <> sum Ns:N_t . (
668                             NEW_NONCE(Ns)
669                             . SEND_EAP_REQ_0(t2, PeerId, req2(PK(sd), Ns))
670                             . UPDATE_NONCE(PeerId, Nonce(Ns))
671                         )
672                         . UPDATE_DATA(PeerId, data)
673                         . UPDATE_TYPE(PeerId, Type(t2))
674                     )
675                     <> (type == t2) -> (
676                         SEND_EAP_FAIL_0
677                         . UPDATE_DATA(PeerId, data)
678                         . UPDATE_STATE(PeerId, s1)
679                         . UPDATE_TYPE(PeerId, Type(no_type))
680                     )
681                     % Waiting Exchange
682                     <> (type == t3) -> (
683                         SEND_EAP_FAIL_0
684                         . UPDATE_STATE(PeerId, s1)
685                         . UPDATE_TYPE(PeerId, Type(no_type))
686                     )
687                     % Completion Exchange
688                     <> (type == t4) -> (
689                         sum pd:DBInterface_t . (
690                             QUERY_DATA(PeerId, pd)
691                             . (MACp(data) != HMAC(
692                                 Kmp(pd), 1, Vers(sd), Verp(pd), PeerId, Cryptosuites(sd), Dirs(sd), ServerInfo(sd),
693                                 Cryptosuitep(pd), Dirp(pd), PeerInfo(pd), PK(sd), Ns(pd), PKp(pd),
694                                 Np(pd), Noob(pd)))
695                             -> (
696                                 SEND_EAP_REQ_0(t0, PeerId, err(E4001))
697                                 . Server_Error(Type(pd), PeerId, E4001)
698                             )
699                             <> SEND_EAP_SUCC_0
700                             . UPDATE_STATE(PeerId, s4)
701                             . UPDATE_TYPE(PeerId, Type(no_type))
702                         )
703                     )
704                     % Reconnect Exchange
705                     <> (type == t5) -> (
706                         sum Ns:N_t . (
707                             NEW_NONCE(Ns)
708                             . UPDATE_NONCE(PeerId, Nonce(Ns))
709                             . sum pd:DBInterface_t . (
710                                 QUERY_DATA(PeerId, pd)
711                                 . (Cryptosuitep(data) == Cryptosuitep(pd)) -> (
712                                     SEND_EAP_REQ_0(t6, PeerId, req6(0, Ns))
713                                 )
714                                 <> SEND_EAP_REQ_0(t6, PeerId, req6(PK2(sd), Ns))
715                             )
716                         )
717                         . UPDATE_DATA(PeerId, data)
718                         . UPDATE_TYPE(PeerId, Type(t6))
719                     )
```

```
720                         <> (type == t6) -> (
721                             ((PKp2(data) != 0) -> (
722                                 sum pd:DBInterface_t . (
723                                     QUERY_DATA(PeerId, pd)
724                                     . UPDATE_KEY(PeerId, Keys(
725                                         RekeyingECDH(192, ECDH(PK2(sd), PKp2(data)), Np2(data), Ns(pd), Kms(pd)),
726                                         RekeyingECDH(224, ECDH(PK2(sd), PKp2(data)), Np2(data), Ns(pd), Kmp(pd))
727                                     ))
728                                 )
729                                 . sum pd:DBInterface_t . (
730                                     QUERY_DATA(PeerId, pd)
731                                     . SEND_EAP_REQ_O(t7, PeerId, req7(
732                                         HMAC(Kms(pd), 2, Vers(sd), Verp(pd), PeerId, Cryptosuites(sd), 0,
733                                         ServerInfo(sd), Cryptosuitep(pd), 0, PeerInfo(pd), PK2(sd), Ns(pd), PKp2(data),
734                                         Np2(data), 0)
735                                     ))
736                                     . UPDATE_TYPE(PeerId, Type(t7))
737                                 )
738                             )
739                             <> (PKp2(data) == 0) -> (
740                                 sum pd:DBInterface_t . (
741                                     QUERY_DATA(PeerId, pd)
742                                     . UPDATE_KEY(PeerId, Keys(
743                                         Rekeying(192, Kms(pd), Np2(data), Ns(pd)),
744                                         Rekeying(224, Kmp(pd), Np2(data), Ns(pd))
745                                     ))
746                                 )
747                                 . sum pd:DBInterface_t . (
748                                     QUERY_DATA(PeerId, pd)
749                                     . SEND_EAP_REQ_O(t7, PeerId, req7(
750                                         HMAC(Kms(pd), 2, Vers(sd), Verp(pd), PeerId, Cryptosuites(sd), 0,
751                                         ServerInfo(sd), Cryptosuitep(pd), 0, PeerInfo(pd), 0, Ns(pd), 0, Np2(data), 0)
752                                     ))
753                                     . UPDATE_TYPE(PeerId, Type(t7))
754                                 )
755                             ))
756                             . UPDATE_DATA(PeerId, data)
757                         )
758                         <> (type == t7) -> (
759                             sum pd:DBInterface_t . (
760                                 QUERY_DATA(PeerId, pd)
761                                 . (
762                                     MACp2(data) != HMAC(
763                                         Kmp(pd), 1, Vers(sd), Verp(pd), PeerId, Cryptosuites(sd), 0, ServerInfo(sd),
764                                         Cryptosuitep(pd), 0, PeerInfo(pd), PK2(sd), Ns(pd), PKp(pd), Np(pd), 0)
765                                     &&
766                                     MACp2(data) != HMAC(
767                                         Kmp(pd), 1, Vers(sd), Verp(pd), PeerId, Cryptosuites(sd), 0, ServerInfo(sd),
768                                         Cryptosuitep(pd), 0, PeerInfo(pd), 0, Ns(pd), 0, Np(pd), 0)
769                                 ) -> (
770                                     SEND_EAP_REQ_O(t0, PeerId, err(E4001))
771                                     . Server_Error(Type(pd), PeerId, E4001)
772                                 )
773                             )
774                             . SEND_EAP_SUCC_O
775                             . UPDATE_STATE(PeerId, s4)
776                             . UPDATE_TYPE(PeerId, Type(no_type))
777                         )
778                         % Completion Exchange
779                         <> (type == t8) -> (
780                             % Invalid OOB message identifier
781                             (!(NoobId(data) in noobs)) -> (
782                                 sum pd:DBInterface_t . (
783                                     QUERY_DATA(PeerId, pd)
784                                     . SEND_EAP_REQ_O(t0, PeerId, err(E1006))
785                                     . Server_Error(Type(pd), PeerId, E1006)
786                                 )
787                             )
788                             <> sum pd:DBInterface_t . (
789                                 QUERY_DATA(PeerId, pd)
790                                 . UPDATE_KEY(PeerId, Keys(
791                                     Completion(192, ECDH(PK(sd), PKp(pd)), Np(pd), Ns(pd), NoobId(data)),
792                                     Completion(224, ECDH(PK(sd), PKp(pd)), Np(pd), Ns(pd), NoobId(data))
793                                 ))
794                             )
795                             . sum pd:DBInterface_t . (
796                                 QUERY_DATA(PeerId, pd)
797                                 . SEND_EAP_REQ_O(t4, PeerId, req4(
798                                     NoobId(data), HMAC(Kms(pd), 2, Vers(sd), Verp(pd), PeerId, Cryptosuites(sd),
799                                     Dirs(sd), ServerInfo(sd), Cryptosuitep(pd), Dirp(pd), PeerInfo(pd), PK(sd), Ns(pd),
800                                     PKp(pd), Np(pd), NoobId(data))
801                                 ))
802                             )
```

```
803                             . UPDATE_NONCE(PeerId, Noob(NoobId(data)))
804                             . UPDATE_TYPE(PeerId, Type(t4))
805                         )
806                     ) . Server(sd, noobs)
807                 )
808             )
809         )
810         % Receive OOB (Peer-to-Server)
811         + (Dirs(sd) == 1 || Dirs(sd) == 3) -> (
812             sum PeerId:Pos, Server_State:State_t, Type:Type_t . (
813                 (PeerId <= max_peers) -> (
814                     QUERY_STATE(PeerId, Server_State, Type)
815                     . (Server_State == s1 && Type == no_type) -> (
816                         sum Noob:Noob_t, Hoob:Hoob_t . (
817                             RECV_OOB_P2S_I(PeerId, Noob, Hoob)
818                             . sum pd:DBInterface_t . (
819                                 QUERY_DATA(PeerId, pd)
820                                 . (Hoob == H(
821                                     Dirp(pd), Vers(sd), Verp(pd), PeerId, Cryptosuites(sd), Dirs(sd), ServerInfo(sd),
822                                     Cryptosuitep(pd), Dirp(pd), PeerInfo(pd), PK(sd), Ns(pd), PKp(pd), Np(pd), Noob)
823                                 ) -> (
824                                     UPDATE_STATE(PeerId, s2)
825                                     . UPDATE_NONCE(PeerId, Noob(Noob))
826                                     . Server(sd, noobs)
827                                 )
828                                 <> sum OobRetries:Nat . (
829                                     QUERY_FAILED_OOBS(PeerId, OobRetries)
830                                     . (OobRetries >= max_oob_retries) -> (
831                                         RESET_DATABASE(PeerId)
832                                         . UPDATE_STATE(PeerId, s0)
833                                     ) <> UPDATE_FAILED_OOBS(PeerId, OobRetries+1)
834                                 ) . Server(sd, noobs)
835                             )
836                         )
837                     ) <> Server(sd, noobs)
838                 )
839             )
840         )
841         % Send OOB (Server-to-Peer)
842         + (Dirs(sd) == 2 || Dirs(sd) == 3) -> (
843             sum PeerId:Pos, Server_State:State_t, Type:Type_t . (
844                 (PeerId <= max_peers) -> (
845                     QUERY_STATE(PeerId, Server_State, Type)
846                     . (Server_State == s1 && Type == no_type) -> (
847                         sum pd:DBInterface_t . (
848                             QUERY_DATA(PeerId, pd)
849                             . sum Noob:Noob_t . (
850                                 NEW_NOOB(Noob)
851                                 . SEND_OOB_S2P_O(PeerId, Noob, H(
852                                     Dirp(pd), Vers(sd), Verp(pd), PeerId, Cryptosuites(sd), Dirs(sd),
853                                     ServerInfo(sd), Cryptosuitep(pd), Dirp(pd), PeerInfo(pd), PK(sd), Ns(pd),
854                                     PKp(pd), Np(pd), Noob)
855                                 )
856                                 . SERV_STATE(PeerId, s1)
857                                 . Server(sd, noobs <| Noob)
858                             )
859                         )
860                     ) <> Server(sd, noobs)
861                 )
862             )
863         )
864     ;
865
866     % Restore the state after sending or receiving an error message
867     %   Type   : Next expected type
868     %   PeerId : Peer ID
869     %   Error  : Error message
870     Server_Error(Type:Type_t, PeerId:PeerId_t, Error:Error_t) =
871         % Alert received error message
872         LOG_ERROR(Error)
873         % Send EAP-Failure and recover from error
874         . SEND_EAP_FAIL_O . (
875             % Error 1006: transition to state 1
876             (Error == E1006) -> (
877                 UPDATE_STATE(PeerId, s1)
878                 . UPDATE_TYPE(PeerId, Type(no_type))
879             )
880             % Error 2002: transition to sink state
881             <> (Error == E2002) -> (
882                 UPDATE_STATE(PeerId, error)
883                 . UPDATE_TYPE(PeerId, Type(no_type))
884             )
885             %% Generic error handling
```

```
886             % Initial Exchange
887             <> (Type == t1 || Type == t2) -> (
888                 UPDATE_STATE(PeerId, s0)
889                 . UPDATE_TYPE(PeerId, Type(no_type))
890             )
891             % Reconnect Exchange
892             <> (Type == t5 || Type == t6 || Type == t7) -> (
893                 UPDATE_STATE(PeerId, s3)
894                 . UPDATE_TYPE(PeerId, Type(no_type))
895             )
896             % Waiting/Completion Exchange
897             <> (Type == t3 || Type == t8 || Type == t4) -> (
898                 UPDATE_TYPE(PeerId, Type(no_type))
899             )
900         )
901     ;
902
903     % EAP Peer
904     Peer(Peer_State:State_t, PeerId:PeerId_t, Type:Type_t, spd:StaticPeerData_t, Vers:Ver_l, Verp:Ver_t,
905         Cryptosuites:Cryptosuite_l, Cryptosuitep:Cryptosuite_t, Dirs:Dir_t, ServerInfo:Info_t, PKs:PK_t, Ns:N_t,
906         Np:N_t, Noob:Noob_t, Dirp:Dir_t, Kmp:K_t, Kms:K_t, noobs:List(Nat), oobretries:Nat) =
907     (!is_error(Peer_State)) -> (
908         % Initial Exchange
909         (Peer_State == s0) -> (
910             (Type == no_type) -> (
911                 SEND_EAP_RES_ID_0(PeerId, Peer_State)
912                 . Peer(Type = id)
913             )
914             + (Type == t2) -> (
915                 RECV_EAP_FAIL_I
916                 . PEER_STATE(PeerId, s1)
917                 . Peer(Peer_State = s1, Type = no_type)
918             )
919         )
920         % Completion/Waiting Exchange
921         + (Peer_State == s1 || Peer_State == s2) -> (
922             (Type == no_type) -> (
923                 SEND_EAP_RES_ID_0(PeerId, Peer_State)
924                 . Peer(Type = id)
925             )
926             % Waiting Exchange
927             + (Type == t3) -> (
928                 RECV_EAP_FAIL_I
929                 . PEER_STATE(PeerId, s1)
930                 . Peer(Peer_State = s1, Type = no_type)
931             )
932             % Completion Exchange
933             + (Type == t4) -> (
934                 RECV_EAP_SUCC_I
935                 . PEER_STATE(PeerId, s4)
936                 . Peer(Peer_State = s4, Type = no_type)
937             )
938         )
939         % Reconnect Exchange
940         + (Peer_State == s3) -> (
941             (Type == no_type) -> (
942                 SEND_EAP_RES_ID_0(PeerId, Peer_State)
943                 . Peer(Type = id)
944             )
945             + (Type == t7) -> (
946                 RECV_EAP_SUCC_I
947                 . PEER_STATE(PeerId, s4)
948                 . Peer(Peer_State = s4, Type = no_type)
949             )
950         )
951         + sum PeerId_RCV:PeerId_t, data:Data_t, type:Type_t . (
952             RECV_EAP_REQ_I(type, PeerId_RCV, data) . (
953                 % Receive error message
954                 (type == t0) -> (
955                     RECV_EAP_FAIL_I . (
956                         % Error 1006: transition to state 1
957                         (ErrorCode(data) == E1006) -> (
958                             PEER_STATE(PeerId, s1)
959                             . Peer(Peer_State = s1, Type = no_type)
960                         )
961                         % Error 2002: transition to sink state
962                         + (ErrorCode(data) == E2002) -> (
963                             PEER_STATE(PeerId, error)
964                             . Peer(Peer_State = error, Type = no_type)
965                         )
966                         % Initial Exchange
967                         + (Type == t1 || Type == t2) -> (
968                             PEER_STATE(PeerId, s0)
```

```
                        . Peer(Peer_State = s0, Type = no_type)
                    )
                    % Reconnect Exchange
                    + (Type == t5 || Type == t6 || Type == t7) -> (
                        PEER_STATE(PeerId, s3)
                        . Peer(Peer_State = s3, Type = no_type)
                    )
                    % Waiting/Completion Exchange
                    + (Type == t3 || Type == t8 || Type == t4) -> (
                        Peer(Type = no_type)
                    )
                    % No exchange
                    + (Type == id || Type == no_type) -> (
                        Peer(Type = no_type)
                    )
                )
            )
            % Invalid data
            <> (is_invalid(data)) -> (
                SEND_EAP_RES_0(t0, PeerId_RCV, err(E1003))
                . RECV_EAP_FAIL_I . (
                    % Initial Exchange
                    (Type == t1 || Type == t2) -> (
                        PEER_STATE(PeerId, s0)
                        . Peer(Peer_State = s0, Type = no_type)
                    )
                    % Reconnect Exchange
                    + (Type == t5 || Type == t6 || Type == t7) -> (
                        PEER_STATE(PeerId, s3)
                        . Peer(Peer_State = s3, Type = no_type)
                    )
                    % Waiting/Completion Exchange
                    + (Type == t3 || Type == t8 || Type == t4) -> (
                        Peer(Type = no_type)
                    )
                    % No exchange
                    + (Type == id || Type == no_type) -> (
                        Peer(Type = no_type)
                    )
                )
            )
            % Unexpected message type
            <> (
                (Type == id && type != t1
                            && type != t3
                            && type != t4
                            && type != t5
                            && type != t8)      ||
                (Type == t1 && type != t2)      ||
                (Type == t2 && type != failure) ||
                (Type == t3 && type != failure) ||
                (Type == t4 && type != success) ||
                (Type == t5 && type != t6)      ||
                (Type == t6 && type != t7)      ||
                (Type == t7 && type != success) ||
                (Type == t8 && type != t4)
            ) -> (
                SEND_EAP_RES(t0, PeerId_RCV, err(E1004))
                . RECV_EAP_FAIL_I . (
                    % Initial Exchange
                    (Type == t1 || Type == t2) -> (
                        PEER_STATE(PeerId, s0)
                        . Peer(Peer_State = s0, Type = no_type)
                    )
                    % Reconnect Exchange
                    + (Type == t5 || Type == t6 || Type == t7) -> (
                        PEER_STATE(PeerId, s3)
                        . Peer(Peer_State = s3, Type = no_type)
                    )
                    % Waiting/Completion Exchange
                    + (Type == t3 || Type == t8 || Type == t4) -> (
                        Peer(Type = no_type)
                    )
                    % No exchange
                    + (Type == id || Type == no_type) -> (
                        Peer(Type = no_type)
                    )
                )
            )
            % Initial Exchange, PeerId not set
            <> (type == t1) -> (
                % No mutually supported protocol version
                (!(exists a,b:Nat .
```

```
1052                        (a < #Vers(data) && b < #Vers(spd) && (Vers(data) . a) == (Vers(spd) . b)))) -> (
1053                            SEND_EAP_RES_0(t0, PeerId_RCV, err(E3001))
1054                            . RECV_EAP_FAIL_I
1055                            . PEER_STATE(PeerId, s0)
1056                            . Peer(Peer_State = s0, Type = no_type)
1057                    )
1058                    % No mutually supported cryptosuite
1059                    <> (!(exists a,b:Nat . (a < #Cryptosuites(data) && b < #Cryptosuites(spd) &&
1060                        (Cryptosuites(data).a) == (Cryptosuites(spd).b)))) -> (
1061                            SEND_EAP_RES_0(t0, PeerId_RCV, err(E3002))
1062                            . RECV_EAP_FAIL_I
1063                            . PEER_STATE(PeerId, s0)
1064                            . Peer(Peer_State = s0, Type = no_type)
1065                    )
1066                    % No mutually supported OOB direction
1067                    <> ((Dirs(data)) != 3 && (Dirs(data)) != Dirs(spd)) -> (
1068                        SEND_EAP_RES_0(t0, PeerId_RCV, err(E3003))
1069                        . RECV_EAP_FAIL_I
1070                        . PEER_STATE(PeerId, s0)
1071                        . Peer(Peer_State = s0, Type = no_type)
1072                    )
1073                    % Invalid server info
1074                    <> (is_invalidinfo(ServerInfo(data))) -> (
1075                        SEND_EAP_RES_0(t0, PeerId_RCV, err(E5002))
1076                        . RECV_EAP_FAIL_I
1077                        . PEER_STATE(PeerId, s0)
1078                        . Peer(Peer_State = s0, Type = no_type)
1079                    )
1080                    % No conflicts, send response back
1081                    <> sum Cryptosuite:Cryptosuite_t, Version:Ver_t, Direction:Dir_t . (
1082                        (Cryptosuite in Cryptosuites(data) && Cryptosuite in Cryptosuites(spd) &&
1083                         Version in Vers(data) && Version in Vers(spd) && Direction == Dirs(spd)
1084                        ) -> (
1085                            SEND_EAP_RES_0(type, PeerId_RCV, res1(Version, Cryptosuite, Direction, PeerInfo(spd)))
1086                        )
1087                        . Peer(PeerId = PeerId_RCV, Type = type, Vers = Vers(data), Verp = Version,
1088                            Cryptosuites = Cryptosuites(data), Cryptosuitep = Cryptosuite, Dirs = Dirs(data),
1089                            Dirp = Direction, ServerInfo = ServerInfo(data)
1090                        )
1091                    )
1092                )
1093                % PeerId set
1094                + (type >= t2) -> (
1095                    % Initial Exchange
1096                    (type == t2) -> (
1097                        sum np:N_t . (
1098                            NEW_NONCE(np)
1099                            . SEND_EAP_RES_0(type, PeerId, res2(PK(spd), np))
1100                            . Peer(Type = type, PKs = PKs(data), Ns = Ns(data), Np = np)
1101                        )
1102                    )
1103                    % Waiting Exchange
1104                    <> (type == t3) -> (
1105                        SEND_EAP_RES_0(type, PeerId, res3)
1106                        . Peer(Type = type)
1107                    )
1108                    % Completion Exchange
1109                    <> (type == t4) -> (
1110                        % Invalid OOB message identifier
1111                        (!(NoobId(data) in noobs) && (NoobId(data) != Noob)) -> (
1112                            SEND_EAP_RES_0(t0, PeerId, err(E1006))
1113                            . RECV_EAP_FAIL_I
1114                            . PEER_STATE(PeerId, s1)
1115                            . Peer(Peer_State = s1, Type = no_type)
1116                        )
1117                        % MAC verification failure
1118                        <> (MACs(data) != HMAC(
1119                            Completion(192, ECDH(PK(spd), PKs), Np, Ns, NoobId(data)),
1120                                2, Vers, Verp, PeerId, Cryptosuites, Dirs, ServerInfo,
1121                                Cryptosuitep, Dirp, PeerInfo(spd), PKs, Ns, PK(spd), Np,
1122                                NoobId(data)
1123                        )) -> (
1124                            SEND_EAP_RES_0(t0, PeerId, err(E4001))
1125                            . RECV_EAP_FAIL_I
1126                            . Peer(Type = no_type)
1127                        )
1128                        <> SEND_EAP_RES_0(type, PeerId, res4(
1129                            HMAC(Completion(224, ECDH(PK(spd), PKs), Np, Ns, NoobId(data)),
1130                                1, Vers, Verp, PeerId, Cryptosuites, Dirs, ServerInfo,
1131                                Cryptosuitep, Dirp, PeerInfo(spd), PKs, Ns, PK(spd), Np,
1132                                NoobId(data))
1133                        ))
1134                        . Peer(Type = type,
```

```
                        Kmp = Completion(224, ECDH(PK(spd), PKs), Np, Ns, NoobId(data)),
                        Kms = Completion(192, ECDH(PK(spd), PKs), Np, Ns, NoobId(data))
                    )
                )
                % Reconnect Exchange
                <> (type == t5) -> (
                    % No mutually supported protocol version
                    (!(exists a,b:Nat .
                        (a < #Vers(data) && b < #Vers(spd) && (Vers(data) . a) == (Vers(spd) . b)))) -> (
                            SEND_EAP_RES_0(t0, PeerId_RCV, err(E3001))
                            . RECV_EAP_FAIL_I
                            . PEER_STATE(PeerId, s0)
                            . Peer(Peer_State = s0, Type = no_type)
                    )
                    % No mutually supported cryptosuite
                    <> (!(exists a,b:Nat .
                        (a < #Cryptosuites(data) && b < #Cryptosuites(spd) &&
                            (Cryptosuites(data).a) == (Cryptosuites(spd).b))) -> (
                                SEND_EAP_RES_0(type, PeerId, err(E3002))
                                . RECV_EAP_FAIL_I
                                . PEER_STATE(PeerId, s3)
                                . Peer(Peer_State = s3, Type = no_type)
                        )
                    )
                    % No conflicts, send response back
                    + sum Cryptosuite:Cryptosuite_t, Version:Ver_t . (
                        (Cryptosuite in Cryptosuites(data) && Cryptosuite in Cryptosuites(spd) &&
                         Version in Vers(data) && Version in Vers(spd)
                        ) -> (
                            SEND_EAP_RES_0(type, PeerId, res5(Version, Cryptosuite))
                            . Peer(Type = type, Vers = Vers(data), Cryptosuitep = Cryptosuite)
                        )
                    )
                )
                <> (type == t6) -> (
                    sum np2:N_t . (
                        NEW_NONCE(np2)
                        . (PKs2(data) == 0) -> (
                            SEND_EAP_RES_0(type, PeerId, res6(0, np2))
                            . Peer(Type = type, Ns = Ns2(data), Np = np2,
                                Kmp = Rekeying(224, Kmp, np2, Ns2(data)),
                                Kms = Rekeying(192, Kms, np2, Ns2(data))
                            )
                        )
                        <> (PKs2(data) != 0) -> (
                            SEND_EAP_RES_0(type, PeerId, res6(PK2(spd), np2))
                            . Peer(Type = type, PKs = PKs2(data), Ns = Ns2(data), Np = np2,
                                Kmp = RekeyingECDH(224, ECDH(PK2(spd), PKs2(data)), np2, Ns2(data), Kmp),
                                Kms = RekeyingECDH(192, ECDH(PK2(spd), PKs2(data)), np2, Ns2(data), Kms)
                            )
                        )
                    )
                )
                <> (type == t7) -> (
                    (MACs2(data) == HMAC(
                        Kms, 2, Vers, Verp, PeerId, Cryptosuites, 0, ServerInfo,
                        Cryptosuitep, 0, PeerInfo(spd), 0, Ns, 0, Np, 0
                    )) -> (
                        SEND_EAP_RES_0(type, PeerId, res7(
                            HMAC(Kmp, 1, Vers, Verp, PeerId, Cryptosuites, 0, ServerInfo,
                            Cryptosuitep, 0, PeerInfo(spd), 0, Ns, 0, Np, 0)
                        ))
                        . Peer(Type = type)
                    )
                    <> (MACs2(data) == HMAC(
                        Kms, 2, Vers, Verp, PeerId, Cryptosuites, 0, ServerInfo,
                        Cryptosuitep, 0, PeerInfo(spd), PKs, Ns, PK2(spd), Np, 0
                    )) -> (
                        SEND_EAP_RES_0(type, PeerId, res7(
                            HMAC(Kmp, 1, Vers, Verp, PeerId, Cryptosuites, 0, ServerInfo,
                            Cryptosuitep, 0, PeerInfo(spd), PKs, Ns, PK2(spd), Np, 0)
                        ))
                        . Peer(Type = type)
                    )
                    <> SEND_EAP_RES_0(t0, PeerId, err(E4001))
                    . RECV_EAP_FAIL_I
                    . PEER_STATE(PeerId, s3)
                    . Peer(Peer_State = s3, Type = no_type)
                )
                % Completion Exchange
                <> (type == t8) -> (
                    SEND_EAP_RES_0(type, PeerId, res8(Noob))
                    . Peer(Type = type)
```

```
                    )
                )
            )
            + (Peer_State == s1 && Type == no_type) -> (
                % Receive OOB (Server-to-Peer)
                (Dirs(spd) == 2 || Dirs(spd) == 3) -> (
                    sum noob:Noob_t, Hoob:Hoob_t . (
                        RECV_OOB_S2P_I(PeerId, noob, Hoob)
                        . (Hoob == H(
                            Dirp, Vers, Verp, PeerId, Cryptosuites, Dirs, ServerInfo,
                            Cryptosuitep, Dirp, PeerInfo(spd), PKs, Ns, PK(spd), Np, noob)
                        ) -> (
                            PEER_STATE(PeerId, s2)
                            . Peer(Peer_State = s2, Noob = noob)
                        )
                        <> (oobretries >= max_oob_retries) -> (
                            PEER_STATE(PeerId, s0)
                            . Peer(Peer_State = s0, oobretries = 0)
                        ) <> Peer(oobretries = oobretries+1)
                    )
                )
                % Send OOB (Peer-to-Server)
                + (Dirs(spd) == 1 || Dirs(spd) == 3) -> (
                    sum noob:Noob_t . (
                        NEW_NOOB(noob)
                        . SEND_OOB_P2S_O(PeerId, noob, H(
                            Dirp, Vers, Verp, PeerId, Cryptosuites, Dirs, ServerInfo,
                            Cryptosuitep, Dirp, PeerInfo(spd), PKs, Ns, PK(spd), Np, noob)
                        )
                        . PEER_STATE(PeerId, s1)
                        . Peer(noobs = noobs <| noob)
                    )
                )
            )
        )
        % Mobility/Timeout/Failure
        + (Peer_State == s4 && Type == no_type) -> (
            PEER_STATE(PeerId, s3)
            . MOBILITY_TIMEOUT_FAILURE
            . Peer(Peer_State = s3)
        )
        % User reset
        + USER_RESET
        . PEER_STATE(PeerId, s0)
        . Peer(Peer_State = s0, PeerId = 0, Type = no_type)
    )
    ;

init
    % Allowed actions
    allow({
        SEND_EAP_RES_ID, RECV_EAP_RES_ID,
        SEND_EAP_REQ, RECV_EAP_REQ,
        SEND_EAP_RES, RECV_EAP_RES,
        SEND_EAP_FAIL, RECV_EAP_FAIL,
        SEND_EAP_SUCC, RECV_EAP_SUCC,
        SEND_OOB_P2S, RECV_OOB_P2S,
        SEND_OOB_S2P, RECV_OOB_S2P,
        MOBILITY_TIMEOUT_FAILURE,
        % USER_RESET,
        SERV_STATE, PEER_STATE,
        LOG_ERROR,
        RNG_MA,
        QUERY_FAILED_OOBS_MA, UPDATE_FAILED_OOBS_MA,
        QUERY_STATE_MA, UPDATE_STATE_MA,
        QUERY_DATA_MA, UPDATE_DATA_MA,
        UPDATE_NONCE_MA, UPDATE_TYPE_MA, UPDATE_KEY_MA,
        RESET_DATABASE_MA,
        MAX_PEERIDS_REACHED,
        MAX_NOOBS_REACHED,
        MAX_NONCES_REACHED
    },
    comm({
        % Send/receive messages
        SEND_EAP_RES_ID_O | SEND_EAP_RES_ID_I -> SEND_EAP_RES_ID,
        RECV_EAP_RES_ID_O | RECV_EAP_RES_ID_I -> RECV_EAP_RES_ID,
        SEND_EAP_REQ_O    | SEND_EAP_REQ_I    -> SEND_EAP_REQ,
        RECV_EAP_REQ_O    | RECV_EAP_REQ_I    -> RECV_EAP_REQ,
        SEND_EAP_RES_O    | SEND_EAP_RES_I    -> SEND_EAP_RES,
        RECV_EAP_RES_O    | RECV_EAP_RES_I    -> RECV_EAP_RES,
        SEND_EAP_FAIL_O   | SEND_EAP_FAIL_I   -> SEND_EAP_FAIL,
        RECV_EAP_FAIL_O   | RECV_EAP_FAIL_I   -> RECV_EAP_FAIL,
        SEND_EAP_SUCC_O   | SEND_EAP_SUCC_I   -> SEND_EAP_SUCC,
```

```
1301          RECV_EAP_SUCC_O   | RECV_EAP_SUCC_I   -> RECV_EAP_SUCC,
1302          SEND_OOB_P2S_O    | SEND_OOB_P2S_I    -> SEND_OOB_P2S,
1303          RECV_OOB_P2S_O    | RECV_OOB_P2S_I    -> RECV_OOB_P2S,
1304          SEND_OOB_S2P_O    | SEND_OOB_S2P_I    -> SEND_OOB_S2P,
1305          RECV_OOB_S2P_O    | RECV_OOB_S2P_I    -> RECV_OOB_S2P,
1306
1307          % Database communication
1308          QUERY_STATE         | QUERY_STATE_DB         -> QUERY_STATE_MA,
1309          QUERY_DATA          | QUERY_DATA_DB          -> QUERY_DATA_MA,
1310          QUERY_FAILED_OOBS   | QUERY_FAILED_OOBS_DB   -> QUERY_FAILED_OOBS_MA,
1311          UPDATE_FAILED_OOBS  | UPDATE_FAILED_OOBS_DB  -> UPDATE_FAILED_OOBS_MA,
1312          UPDATE_STATE        | UPDATE_STATE_DB        -> UPDATE_STATE_MA,
1313          UPDATE_DATA         | UPDATE_DATA_DB         -> UPDATE_DATA_MA,
1314          UPDATE_NONCE        | UPDATE_NONCE_DB        -> UPDATE_NONCE_MA,
1315          UPDATE_TYPE         | UPDATE_TYPE_DB         -> UPDATE_TYPE_MA,
1316          UPDATE_KEY          | UPDATE_KEY_DB          -> UPDATE_KEY_MA,
1317
1318          % Reset database for a given PeerId
1319          RESET_DATABASE | RESET_DATABASE_DB -> RESET_DATABASE_MA,
1320
1321          % Random value generation
1322          NEW_NONCE_RNG  | NEW_NONCE  -> RNG_MA,
1323          NEW_NOOB_RNG   | NEW_NOOB   -> RNG_MA,
1324          NEW_PEERID_RNG | NEW_PEERID -> RNG_MA
1325      },
1326
1327      % Start processes
1328      Server(static_serv_data, []) ||
1329      Peer(s0, 0, no_type, static_peer_data, [], 0, [], 0, 0, no_info, 0, 0, 0, 0, 0, no_key, no_key, [], 0) ||
1330      Rng(1, 1, 1) ||
1331      Database(state, verp, cryptosuitep, dirp, peerinfo, pkp, np, ns, noob, oobretries, type, kms, kmp) ||
1332      ServerToPeerChannel(true) || PeerToServerChannel(false) ||
1333      ServerToPeerOOBChannel || PeerToServerOOBChannel || ServerToPeerOOBChannel || PeerToServerOOBChannel
1334      ));
```